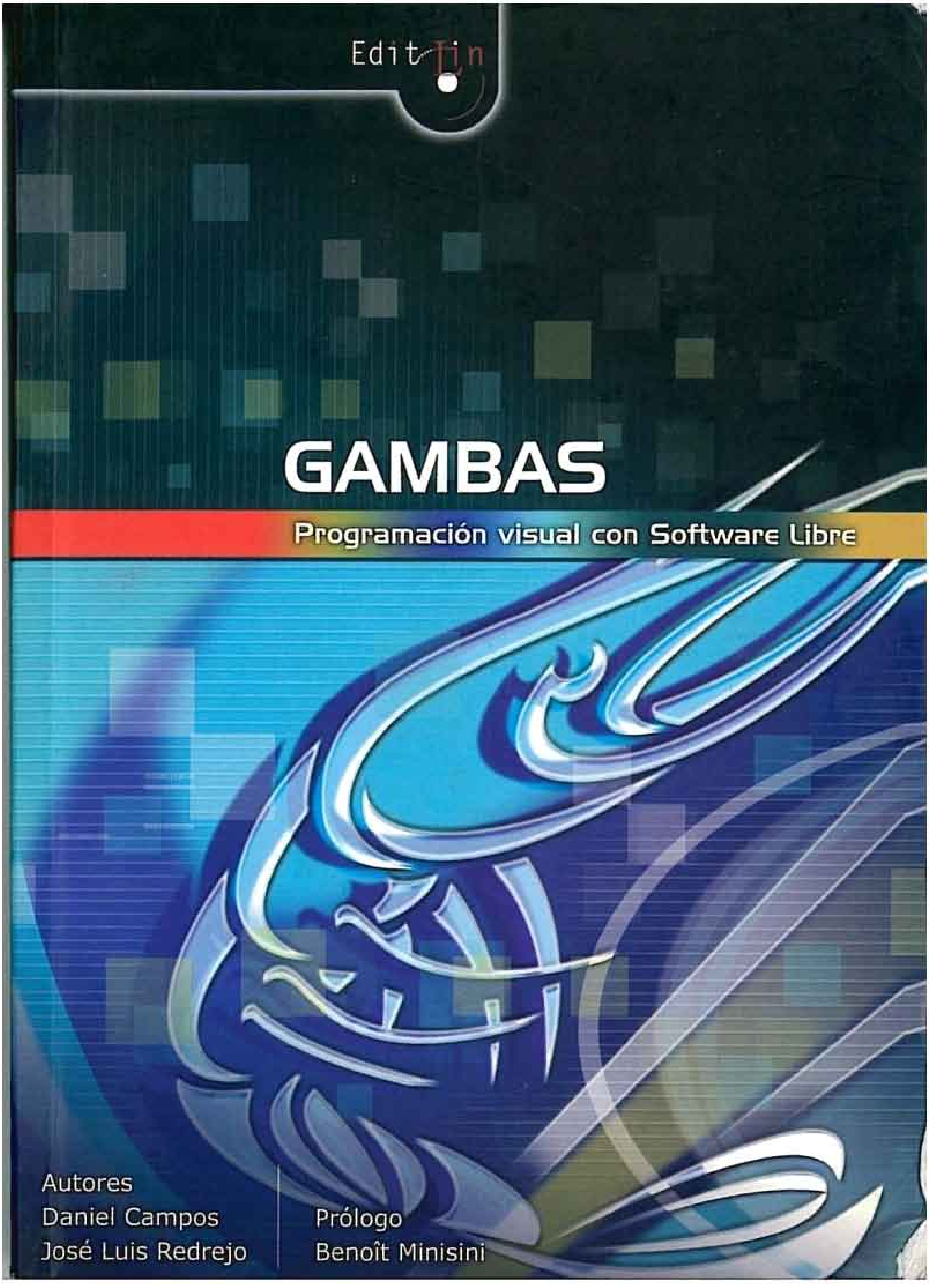

Edit Lin
GAMBAS
Programación visual con Software Libre
Autores
Daniel Campos
José Luis Redrejo
Prólogo
Benoît Minisini

Comunidad WWW.GAMBAS-ES.ORG

GAMBAS

Gambas Almost Means Basic

Comunidad WWW.GAMBAS-ES.ORG

INFORMACIÓN

Tienes en tus manos el libro COMPLETO de "GAMBAS programación visual con Software Libre" escrito por Daniel Campos y José Luis Redrejo bajo licencia libre.

Al libro electrónico original le faltan las páginas 34, 35, 58, 59, 104, 105, 166, 167, 190, 191, 216, 260 y 261.

Hemos escaneado e insertado estas páginas faltantes en este nuevo libro electrónico, creando así el libro electrónico completo que todo "gambero" debería leer.

Hemos modificado el libro original, añadido las páginas faltantes, unificado todos los capítulos en un único fichero y lo volvemos a hacer público conservando la misma licencia libre.

Comunidad WWW.GAMBAS-ES.ORG

06/01/2010

Autores:
DANIEL CAMPOS
JOSÉ LUIS REDREJO

Responsable editorial:
SORAYA MUÑOZ

Responsable de Marketing y Comunicación:
ÁLVARO GARCÍA

Edición:
MIRIAM MONTERO

Diseño de portada:
PICTOGRAMA

Maquetación:
ÁNGEL ESTRADA

ISBN: 978-84-934371-2-3
Depósito Legal: M-41116-2005

Edita:
©EDIT LIN EDITORIAL, S.L., 2005
Avda. Portugal, 85- local
28011 Madrid (España)
Tel.: 91 577 03 55
Fax: 91 577 06 18

www.librosdelinux.com
info@librosdelinux.com

Impreso en España (Printed in Spain)

*Este libro ha sido realizado con Software Libre, concretamente con: OpenOffice.org, Evolution, Mozilla, GIMP.

PRÓLOGO

La escena se desarrolla en algún lugar en el campo, en verano...

– ¡Abuelo, no te duermas! No has acabado de contarme la historia del ordenador...

– ¿Eh?... ah... sí... bueno, bueno...

El abuelo bostezó y pensó en lo que iba a decir. Ya había anochecido y sólo se oía el canto de los grillos y el rechinar de la butaca.

"Cuando mi hermano compró su primer ordenador personal, estaba compuesto de un lector de cintas, de algunos KB de memoria -no recuerdo exactamente cuántos- y de un teclado mecánico, cosa poco frecuente en aquella época. Pero, sobre todo, disponía de un lenguaje BASIC completo y de un manual que intentaba enseñar la programación paso a paso.

Algunos años más tarde fui yo el que me compré un ordenador. Los modelos eran más potentes, pero volví a encontrar el lenguaje BASIC y la posibilidad de sacar el máximo rendimiento.

En aquella época, las revistas hablaban de programación, la gente programaba e intercambiaba programas. Algunos copiaban programas que no habían creado y se les llamaba piratas... No, no, todos no tenían un parche negro en un ojo y un garfio en lugar de una mano.

Después llegó la época del IBM PC y poco a poco las cosas empezaron a cambiar. Al principio era una enorme caja gris, fea y ruidosa. Pero aumentó en potencia y mientras más potencia adquiría, menos posibilidad había de sacarle el máximo rendimiento.

Gracias a un puñado de futuros millonarios teníamos pantallas más grandes, más colores, más sonido, más memoria, más capacidad de almacenamiento. Pero, al mismo tiempo, se chapuceaban los sistemas operativos, se despreciaban los principios elementales de seguridad, se frenaban las aplicaciones, se bloqueaban los formatos de datos, se ocultaba y se mentía sobre el funcionamiento interno del conjunto.

El usuario se convertía en un consumidor como cualquier otro. Campañas de publicidad gigantescas le conducían a comprar una especie de caja negra de la que seguidamente había que controlar al máximo la utilización. A pesar de todo, la gente continuaba intercambiando y los piratas pirateando.

En cuanto a los que realmente eran capaces de sacar el máximo rendimiento a los ordenadores, tenían que comprarlos y después encerrarlos tras las puertas de un campus o de una cláusula de no divulgación. Si no hacían esto, tenían que dejar de programar.

Afortunadamente, las cosas no sucedieron exactamente como estaba previsto y gracias al nacimiento de Internet, y a la mezcla de talentos de numerosos programadores, a menudo voluntarios, idealistas y pragmáticos al mismo tiempo, se impuso un movimiento de liberación y nació, entre otros, el sistema operativo GNU/Linux.

Pero esto es otra historia, te la contaré mañana..."

La idea de desarrollar Gambas sobre Linux en casa vino principalmente como reacción a la obligación de utilizar Visual Basic sobre Windows en el trabajo.

Cualquier usuario con un poco de idea, cualquier empresa, cualquier administración, necesitan sacar el mayor rendimiento posible a su ordenador, es decir, hacer de

él lo que quieran hacer sin tener las competencias necesarias requeridas por los lenguajes de programación, ni el tiempo para adquirirlas.

Con un lenguaje incoherente, incompleto y repleto de fallos, Visual Basic ha respondido, a pesar de todo, a estas necesidades. Pero sus creadores, encerrados en su torre de marfil, lo han ido poco a poco abandonando. ¿Había quizás desprecio hacia esos usuarios que tenían la pretensión de sacar, como a ellos les pareciera, el máximo rendimiento de su herramienta de trabajo?

A pesar de que forma parte de la gran familia BASIC y que la interfaz de los programas se dibuja con el ratón, Gambas no tiene ningún otro punto en común con Visual Basic y es totalmente incompatible con él. Sin embargo, mi prioridad es que pueda satisfacer la misma necesidad: conseguir de la manera más simple posible sacar el máximo rendimiento de todas las funciones del sistema operativo subyacente, GNU/Linux en este caso preciso.

Gambas se inspira, sobre todo, en Java: se trata de un lenguaje orientado a objetos, procedural e interpretado. Pero su intérprete es mucho más rudimentario: no hay compilación *just-in-time*, no hay *garbage collector*. Sin embargo, he cuidado mucho su simplicidad, su coherencia, su fiabilidad y sus facultades de internacionalización.

Gambas es extensible. Es posible escribir componentes que añaden al lenguaje clases suplementarias. Estos componentes ofrecen, en general, el acceso a librerías específicas: QT, GTK+, SDL, OpenGL, y así sucesivamente...

Gambas ofrece un entorno de desarrollo integrado, moderno, que está, él mismo, escrito en Gambas. Es ése principalmente el que me permite probar el lenguaje :-). Además, es totalmente posible prescindir de él.

Finalmente, y nunca insistiré lo suficiente, Gambas no es un lenguaje creado por programadores profesionales para programadores no profesionales. Yo soy su primer usuario y otra de las razones que me ha llevado a crearlo es que todavía no deseo aprender en serio Perl o Python :-)

La primera versión estable de Gambas salió el 1 de enero de 2005, pero las cosas se pondrán realmente interesantes con la segunda versión. En el momento en el que escribo estas líneas, la versión de desarrollo ofrece la posibilidad de escribir componentes directamente en Gambas, sin necesidad de dominar C o C++. Más tarde, esta posibilidad permitirá al entorno de desarrollo diseñar algo más que formularios gráficos: tablas, hojas HTML, etc.

Es difícil prever de manera precisa cuándo saldrá esta segunda versión, e incluso, más en general, cómo evolucionará Gambas en el futuro. El trabajo que queda por hacer es todavía enorme y las cosas dependen mucho de los usuarios.

En cualquier caso, espero que tendréis tanto gusto en utilizarlo como yo lo he tenido en crearlo y que participaréis en su evolución. Y si, como para otros, Gambas es el factor que os lleva a utilizar por fin el Software Libre, espero que sentiréis como yo el respiro que produce esta libertad.

Martes, 16 de agosto de 2005.

— BENOÎT.

ÍNDICE

CAPÍTULO 1: ¿QUÉ ES GAMBAS? 17

1.1 El lenguaje BASIC: su historia 18
1.2 Un entorno libre 21
1.3 Elementos de Gambas 23
1.4 Cómo obtenerlo 24
1.5 Compilación y dependencias 26
1.6 Familiarizarse con el IDE 27
El primer ejemplo 29
Mejor con un ejemplo gráfico 32
Un poco de magia 35
1.7 Sistema de componentes 36

CAPÍTULO 2: PROGRAMACIÓN BÁSICA 41

2.1 Organización de un proyecto de Gambas 42
Declaración de variables 42
Subrutinas y funciones 45

2.2 Tipos de datos 49
Conversión de tipos 50
Matrices 52
2.3 Operaciones matemáticas 54
Operaciones lógicas 56
2.4 Manejo de cadenas 56
2.5 Control de flujo 60
IF... THEN... ELSE 60
Select 62
FOR 63
WHILE y REPEAT 64
Depuración en el IDE de Gambas 66
2.6 Entrada y salida de ficheros 68
2.7 Control de errores 72
2.8 Programación orientada a objetos con Gambas 73
2.9 Propiedades, Métodos y Eventos 79

CAPÍTULO 3: LA INTERFAZ GRÁFICA 81

3.1 Concepto 81
Partiendo de la consola 84
El entorno de desarrollo 85
3.2 Manejo básico de los controles 87
Posición y tamaño 87
Visibilidad 89
Textos relacionados 89
Colores 90
Ratón 92
Teclado 95
3.3 Galería de controles 96
Controles básicos 96

Otros controles básicos misceláneos 99
Listas de datos 100
Otros controles avanzados 101
3.4 Diálogos 101
La clase Message 101
La clase Dialog 104
Diálogos personalizados 106
3.5 Menús 111
3.6 Alineación de los controles 114
Propiedades de la alineación 114
Controles con alineación predefinida 117
Diseño de una aplicación que aprovecha
este recurso 117
3.7 Introducción al dibujo de primitivas 120

CAPÍTULO 4: GESTIÓN DE PROCESOS 125

4.1 La ayuda ofrecida por otros programas 125
4.2 Gestión potente de procesos 126
4.3 EXEC 127
Palabra clave WAIT 128
El descriptor del proceso 130
Redirección con TO 133
Matar un proceso 134
Redirección de la salida estándar
de errores 136
Redirección de la salida estándar 139
Evento Kill() y la propiedad Value 142
Redirección de la entrada estándar,
el uso de CLOSE 146
Notas finales sobre el objeto Process 148
4.4 SHELL 148

CAPÍTULO 5: GESTIÓN DE BASES DE DATOS 151

5.1 Sistemas de bases de datos 151
5.2 Bases de datos y Gambas 153
5.3 Gambas-database-manager, el gestor gráfico 154
Crear una base 154
Crear una tabla 158
Gestionar datos de una tabla 164
SQL 165
5.4 Programación 167
Modelo de bases de datos 167
Conectándose por código 168
Consulta de datos 171
Borrar registros 174
Añadir registros 176
Modificar registros 180
5.5 Otras características 190
Estructura de las tablas 190
Más utilidades del Gestor de Bases de Datos 192

CAPÍTULO 6: RED 195

6.1 Conceptos 195
6.2 Creando un servidor TCP 198
6.3 Un cliente TCP 205
6.4 Clientes y servidores locales 211
6.5 UDP 213
6.6 Resolución de nombres 216
6.7 Protocolo HTTP 220
6.8 Protocolo FTP 225

CAPÍTULO 7: XML 227

7.1 Escritura con XmlWriter 230
7.2 Lectura con XmlReader 238
Modelos de lectura 238
Planteamiento inicial 238
Un ejemplo de lectura 240
7.3 XSLT 252
¿Qué es XSLT? 252
Una plantilla de ejemplo 252
Transformando el documento con Gambas 254
7.4 Acerca de XML-RPC 255

CAPÍTULO 8: HERENCIA 257

8.1 Lenguajes orientados a objetos y herencia 257
8.2 Conceptos necesarios 258
La clase padre 258
La clase hija. Palabra clave INHERITS 262
Extendiendo funcionalidades. Palabra clave SUPER 264
Modificando funcionalidades 265
Reemplazando métodos especiales: _New y _Free 269
Limitaciones 272
8.3 Crear un nuevo control con Gambas 273
Planteando el código 273
Implementación básica 274
8.4 Nuevos componentes con Gambas 279
Preparación del código. Palabra clave EXPORT 280
Archivos necesarios, ubicaciones 280
Probando el nuevo componente 282

CAPÍTULO 9: ACCESO A LA API 287

9.1 Declarar una función externa 288
Cómo denominar la librería 289
El uso de los alias 290
Tipos de datos 291
9.2 Funciones auxiliares 291
9.3 Un ejemplo con libaspell 293
9.4 Obtener información acerca de la librería 297
9.5 Resumen 300

APÉNDICE: Marcas registradas 301

I. ¿Qué es Gambas?

I. ¿Qué es Gambas?
I. ¿Qué es Gambas?
I. ¿Qué es Gambas?

Es obvio que para los que hablamos el idioma español, la palabra gambas nos sugiere algunas cosas, pero todas ellas bastante alejadas del mundo de los ordenadores en general y del entorno de la programación en particular. El autor original de Gambas, Benoît Minisini, no habla una palabra de nuestro idioma y su inocente ignorancia le condujo a nominar su obra con el acrónimo GAMBAS: *Gambas almost means BASIC*, es decir, 'Gambas casi quiere decir BASIC'. No es la primera vez que un nombre o una marca bautizados en otros idiomas produce estos extraños cambalaches, recordemos que Coca Cola(TM) tuvo que cambiar su pronunciación en China porque la primera versión de su nombre significaba *muerde el renacuajo de cera*, o algunos todavía recordamos nuestra cara de asombro al ver los anuncios de un coche al que Suzuki dio en llamar *Pajero*. Tampoco está mal recordar que ya había antecedentes de otros lenguajes de programación con nombre de animal, como Camel

o Python, aunque en esos casos el nombre estaba en inglés y en español no resultaba tan chocante.

En fin, como Benoît, que tiene los derechos de autor, no desea cambiar el nombre, nos tendremos que ir acostumbrando a que Gambas empiece a sonarnos a algo más que a buen marisco. De hecho, Gambas abre el entorno de la programación visual en Linux a todo el mundo, como lo hizo en su día Visual Basic(TM) en Windows. Pero como el tiempo no pasa en vano, Gambas intenta no reproducir los errores que se cometieron entonces. La ampliación del lenguaje BASIC alcanza con Gambas amplias cotas de potencia, profesionalidad y modernidad, sin abandonar nunca la sencillez y claridad de este lenguaje de programación de alto nivel. Ya nunca más se podrá decir que construir aplicaciones visuales para Linux es un proceso largo y complejo que lleva años de trabajo a gurús y maniáticos de la informática.

Gambas no es sólo un lenguaje de programación, es también un entorno de *programación visual* para desarrollar aplicaciones gráficas o de consola. Hace posible el desarrollo de aplicaciones complicadas muy rápidamente. El programador diseña las ventanas de forma gráfica, arrastra objetos desde la **Caja de Herramientas** y escribe código en BASIC para cada objeto. Gambas está *orientado a eventos,* lo que significa que llama automáticamente a los procedimientos cuando el usuario de la aplicación elige un menú, hace clic con el ratón, mueve objetos en la pantalla, etc.

1.1 El lenguaje BASIC: su historia

El nombre BASIC corresponde a las siglas *Beginner's All Purpose Symbolic Instruction Code* (Código para principiantes de instrucciones simbólicas con cualquier propósito). El lenguaje fue desarrollado en 1964 en el Dartmouth College por los matemáticos John George Kemeny y Tom Kurtzas. Intentaban construir un lenguaje de programación fácil de aprender para sus estudiantes de licenciatura. Debía ser un paso intermedio antes de aprender otros más potentes de aquella

época, como FORTRAN o ALGOL. Este último era el lenguaje más utilizado en aplicaciones de procesos de datos, mientras que FORTRAN era empleado en las aplicaciones científicas. Sin embargo, ambos eran difíciles de aprender, tenían gran cantidad de reglas en las estructuras de los programas y su sintaxis. El primer programa hecho en BASIC se ejecutó a las 4 de la madrugada del 1 de mayo de 1964. Debido a su sencillez, BASIC se hizo inmediatamente muy popular y se empezó a usar tanto en aplicaciones científicas como comerciales. Tuvo el mismo impacto en los lenguajes de programación que la aparición del PC sobre los grandes ordenadores.

Cuando se desarrolló BASIC eran los tiempos en los que la informática estaba recluida en universidades y grandes empresas, con ordenadores del tamaño de una habitación. Pero pronto las cosas empezaron a cambiar. En 1971 Intel fabricaba el primer microprocesador. En 1975, la empresa MITS lanzó al mercado un kit de ordenador llamado Altair 8800 a un precio de 397 dólares. Era un ordenador barato, pero no para gente inexperta, había que saber electrónica para montarlo. Además tenía sólo 256 bytes (no es una errata, solo bytes, nada de Kbytes, megas o gigas) y se programaba en código máquina a base de 0 y 1, moviendo unos interruptores que tenía en el frontal. Dos jovencitos vieron un modelo en una revista de electrónica y decidieron montarlo. Le ofrecieron al dueño de MITS, además, hacer un intérprete de BASIC para los nuevos modelos de Altair. Eran William Gates y Paul Allen, y aquel BASIC, con un tamaño de 4 Kbytes, fue el primer producto que entregó una nueva empresa llamada Microsoft. Fue sólo el principio. A finales de los 70, Allen y Gates habían portado BASIC ya a un buen número de plataformas: Atari, Apple, Commodore... Y en 1981, cuando desarrollaron DOS para IBM y su nuevo PC, añadieron también su propio intérprete de BASIC al sistema. En posteriores años siguieron otras versiones hechas por otras compañías como Borland, pero el declive de BASIC había empezado. Las interfaces gráficas de ventanas que Apple popularizó y Microsoft adoptó con sucesivas versiones de



Figura 1. Lanzamiento del Altair 8800.

Windows(TM), se convirtieron en un estándar y BASIC no era un lenguaje preparado para estos entornos.

Sin embargo, en marzo de 1988, un desarrollador de software llamado Alan Cooper[1] intentaba vender una aplicación que permitía personalizar fácilmente el entorno de ventanas usando el ratón. El programa se llamaba Tripod y en aquellas fechas consiguió que William Gates lo viera y le encargara el desarrollo de una nueva versión a la que llamaron Ruby, y a la que añadieron un pequeño lenguaje de programación. Microsoft reemplazó ese lenguaje por su propia versión de BASIC, Quickbasic y el 20 de marzo de 1991 se lanzó al mercado con el nombre de Visual Basic. Al principio fue un verdadero fracaso de ventas, pero la versión 3 publicada en el otoño de 1996 fue un éxito total, tanto que actualmente es el lenguaje de programación más usado. Visual Basic siguió evolucionando hasta la versión 6.0. En 2002 fue integrado en la plataforma .NET de desarrollo, en lo que para muchos de los seguidores ha supuesto el abandono de Microsoft, ya que ha cambiado buena parte de la sintaxis añadiéndole complejidad en contradicción con el espíritu y el nombre del lenguaje. En cualquier caso, a día de hoy se calcula que entre el 70% y 80% de todas las aplicaciones desarrolladas en Windows se han hecho con alguna de las versiones de Visual Basic.

Las causas del éxito de Visual Basic son numerosas, pero entre otras se puede señalar como obvia el uso del lenguaje BASIC que fue pensado para un aprendizaje fácil. Otro de los motivos es disponer de un entorno de desarrollo cómodo, que hace un juego de niños el diseño de la interfaz gráfica de cualquier aplicación, apartando al programador de perder tiempo en escribir el código necesario para crear ventanas, botones, etc., y dejándole centrarse únicamente en la solución al problema que cualquier programa intenta resolver. Con la popularización de sistemas operativos libres como GNU/Linux, éstas y otras razones hacían prever que la aparición de un entorno equivalente libre sería un éxito y contribuiría a la presentación de muchos nuevos desarrollos que lo utilizarían. Ha habido varios intentos que no han cuajado, bien por la lentitud de su evolución, bien por su dificultad de uso o por no ser totalmente libres y no haber arrastrado a una comunidad de usuarios detrás. Finalmente, Benoît Minisini, un programador con experiencia en la escritura de compiladores

que estaba harto de luchar contra los fallos de diseño de Visual Basic, y deseaba poder usar un entorno de GNU/Linux fácil en su distribución, comenzó a desarrollar su propio entorno para Linux basado en BASIC. El 28 de febrero de 2002 puso en Internet la primera versión pública de Gambas: gambas 0.20. Benoît eliminó del diseño del lenguaje bastantes de los problemas que Visual Basic tenía, como la gestión de errores, y le añadió características comunes en los lenguajes actuales más modernos, como la orientación a objetos y la propia estructura de los programas[2]. Como prueba de fuego, el propio entorno de desarrollo fue programado en Gambas desde la primera versión, sirviendo a un tiempo de demostración de la potencia del lenguaje y de detección de necesidades y corrección de errores que se fueron incorporando a las distintas versiones.

En enero de 2005, Benoît publicó la versión 1.0, en la que ya se incorporaba un puñado de componentes desarrollados por otros programadores que colaboraron con él: Daniel Campos, Nigel Gerrard, Laurent Carlier, Rob Kudla y Ahmad Kahmal. Esta versión se consideró suficientemente estable y cerró un ciclo. A partir de esa fecha empezó la programación de la versión 2.0. Ésta ya incluye algunas mejoras en el lenguaje, muchos más componentes y un nuevo modelo de objetos que permitirán usar Gambas en un futuro para el desarrollo de aplicaciones web con la misma filosofía y facilidad que actualmente se usa para aplicaciones de escritorio.

1. 2 Un entorno libre

Gambas es un entorno de desarrollo que se distribuye con la licencia GPL GNU (*General Public Licence*[3]). Esto significa que se distribuye siempre con el código fuente y respeta las cuatro libertades que define la Free Software Foundation:

- La libertad de usar el programa con cualquier propósito (libertad 0).

- La libertad de estudiar cómo funciona el programa y adaptarlo a las propias necesidades (libertad 1). El acceso al código fuente es una condición previa para esto.

- La libertad de distribuir copias, con las que se puede ayudar al vecino (libertad 2).

- La libertad de mejorar el programa y hacer públicas las mejoras a los demás, de modo que toda la comunidad se beneficie (libertad 3). El acceso al código fuente es un requisito previo para esto.

Una de los engaños más comunes en el uso de Software Libre es la creencia de que este modelo de desarrollo obliga a que el trabajo se publique gratis, lo que es del todo incierto. Estas cuatro libertades permiten que, quien lo desee, venda copias de Gambas (entregando siempre el código fuente y respetando esas cuatro libertades) y, por supuesto, de cualquier aplicación desarrollada con este programa. Las aplicaciones desarrolladas con Gambas pueden o no acogerse a la licencia GPL.

También cualquier programador es libre de alterar el propio lenguaje y modificarlo a su gusto, siempre y cuando entregue el código correspondiente a esas modificaciones y respete los derechos de autor de los desarrolladores originales.

Aparte de estas libertades propias de la naturaleza de un proyecto de Software Libre sobre GNU/Linux, Gambas añade más facilidades para el programador:

- Una ayuda muy completa del lenguaje y cada uno de los componentes, algo que es muy de agradecer para los que empiezan a programar en Gambas, y que no es habitual en los proyectos de Software Libre. La ayuda que se publica está en inglés, pero existe un grupo de personas trabajando en la traducción a español[4]. Si todos nos animamos a colaborar[5] en la traducción, pronto estará completa y disponible para el resto de usuarios.

- Una API (Interfaz para programar la aplicación) sencilla y bien documentada, lo que facilita a los programadores crear nuevos componentes para Gambas. La API no es de utilidad inmediata para quien desarrolle con este lenguaje, pero permite a los programadores avanzados que lo deseen añadir funcionalidades al entorno de desarrollo y crear nuevas herramientas para Gambas.

El lenguaje está preparado para ser independiente del gestor de ventanas que use. Esto significa que, sin cambiar una sola línea de código, una aplicación puede ser compilada para ser ejecutada en un escritorio Gnome o KDE, usando las librerías propias de ese escritorio y siendo una aplicación nativa de ese entorno. En el futuro se pueden desarrollar componentes para Windows(TM), Fluxbox y otros gestores de ventanas, y los programas no tendrán que modificar su código para que sean aplicaciones nativas de esos entornos. Marcando, antes de compilar, una opción en el entorno de desarrollo para elegir el componente a usar (actualmente se puede elegir entre *gtk* y *qt*, para Gnome o KDE), se generan distintas aplicaciones para distintos entornos con el mismo código fuente. Esta característica no se encuentra disponible en ningún otro lenguaje existente, lo que convierte a Gambas en un entorno único.

1. 3 Elementos de Gambas

Para poder desarrollar y ejecutar programas hechos con Gambas, son necesarios distintos elementos:

- Un **compilador**, que se encargará de transformar todo el código fuente y archivos que formen parte de un proyecto hecho en Gambas, en un programa ejecutable.

- Un **intérprete** capaz de hacer que los programas hechos en Gambas sean ejecutados por el sistema operativo.

- Un **entorno de desarrollo** que facilite la programación y diseño de las interfaces gráficas de los programas.

- **Componentes** que añaden funcionalidades al lenguaje. La palabra *componente* en Gambas tiene un significado específico, ya que no alude a partes genéricas, sino a librerías específicas que le dotan de más posibilidades. En la actualidad existen componentes para usar xml, conexiones de red, opengl, sdl, ODBC, distintas bases de datos, expresiones regulares, escritorios basados en *qt*, en *gtk*,

etc. Estos componentes son desarrollados por distintos programadores, siguiendo las directrices de la API de Gambas y la documentación publicada al efecto por Benoît Minisini.

I. 4 Cómo obtenerlo

Las nuevas versiones de Gambas se publican a través de la página web oficial del proyecto: **http://gambas.sourceforge.net**. En la actualidad existen dos ramas de Gambas: la rama estable o 1.0 y la rama de desarrollo o 1.9 que desembocará en la versión 2.0. De la rama estable sólo se publican nuevas versiones cuando es para corregir algún fallo que se haya descubierto.

En el momento de escribir estas líneas, la versión estable era la 1.0.11 y no se prevé que haya cambios en el futuro. La versión de desarrollo está en continuo cambio, mejora y adición de nuevos componentes. Esto la hace muy atractiva, debido a la existencia de importantes funcionalidades que no están en la versión estable (como los componentes gtk y ODBC). Sin embargo, al ser una rama en desarrollo, es muy probable que tenga fallos no descubiertos y es seguro que sufrirá cambios en un futuro próximo. En este texto trataremos todas las novedades que la versión de desarrollo contiene para que sea el usuario el que escoja con qué rama trabajar. Buena parte de las diferencias se encuentran en los nuevos componentes. Algunos de estos serán tratados en este texto, por lo que si el lector quiere trabajar con ellos deberá usar la rama de desarrollo.

Las nuevas versiones se publican siempre en forma de código fuente, para que los usuarios que lo deseen compilen el código y obtengan todas las partes que Gambas tiene. Los autores de algunos de los componentes que se han hecho para Gambas, publican de forma separada en distintos sitios web las versiones nuevas de estos, pero todas se envían a Benoît Minisini y pasan a formar parte de la publicación completa de este lenguaje de programación en la siguiente versión. De este modo, se puede decir que cuando Benoît hace pública una nueva, el paquete del código fuente contiene las últimas versiones de todo el conjunto en ese momento.

Como la compilación de Gambas y todos los componentes asociados puede ser una tarea difícil para usuarios no expertos, es común que se creen paquetes binarios con la compilación ya hecha y listos para ser instalados en distintas distribuciones de gnu/Linux. En la misma página web donde se puede bajar el código fuente se encuentran los enlaces para la descarga de los paquetes compilados para estas distribuciones. Existen actualmente paquetes disponibles para Debian, Fedora, Mandriva, Gentoo, Slackware, QiLinux y Suse. En algunos casos, como para Fedora y Debian, están disponibles tanto los paquetes de la versión estable como la de desarrollo.

En el caso de Debian, los paquetes son realizados en gnuLinEx y, posteriormente, subidos a Debian para que estén disponibles y usables en esta distribución y en todas sus derivadas, como Knoppix, Guadalinex, Progeny, Xandros, Linspire, Skolelinux, etc. Por este motivo, las últimas versiones están siempre disponibles antes en los repositorios de gnuLinEx hasta que son subidos y aprobados en Debian. Las líneas del archivo */etc/apt/sources.list* de un sistema Debian para instalar la versión más actualizada de los paquetes de Gambas son:

Para la versión estable:

```
deb http://apt.linex.org/linex  gambas/
```

Para la versión de desarrollo[6]:

```
deb http://www.linex.org/sources/linex/debian/ cl gambas
```

A continuación, en cualquiera de los dos casos, para instalar todos los paquetes de Gambas, hay que ejecutar como usuario root:

```
apt-get update
apt-get install gambas
```

Aunque el código fuente de Gambas se distribuye en un único archivo comprimido, la instalación desde paquetes compilados se hace con un buen puñado de archivos.

La razón es que no todos son necesarios para ejecutar aplicaciones hechas en Gambas. En las distribuciones de Linux se ha seguido el criterio de separar en distintos paquetes el entorno de desarrollo (paquete *gambas-ide*), el intérprete (paquete *gambas-runtime*), y se ha hecho un paquete separado para cada uno de los componentes. Si se quiere programar en Gambas son necesarios la mayoría de ellos, al menos los que el entorno de desarrollo necesita. Si se quiere ejecutar un programa hecho con este lenguaje, sólo es necesario *gambas-runtime* y un paquete por cada uno de los componentes que el programa use. Por ejemplo, si es un programa que está hecho para el escritorio Gnome y no usa ningún otro componente, sólo sería necesario instalar en el sistema los paquetes *gambas-runtime* y *gambas-gb-gtk*.

1. 5 Compilación y dependencias

Si en lugar de instalar paquetes ya compilados para la distribución de gnu/Linux deseamos compilar Gambas desde el código fuente, deberemos seguir los pasos habituales en los sistemas GNU. Es decir, descomprimir el archivo con las fuentes y, desde el directorio que se crea al descomprimir y usando un terminal, ejecutar las siguientes instrucciones:

```
./configure
make
make install
```

La última de ellas debemos hacerla como root, si queremos que el programa esté disponible para todos los usuarios del ordenador. Si estamos habituados a compilar aplicaciones en sistemas GNU, disponemos ya de un compilador instalado y de bastantes librerías de desarrollo. Las instrucciones anteriores tratarán de compilar e instalar todos los componentes de Gambas, que son muchos. Si no tenemos las librerías correspondientes a alguno de ellos, simplemente no se compilarán y la instrucción **./configure** nos informará de ello. Es importante saber que el entorno de desarrollo está hecho sobre las librerías gráficas *qt*, por tanto, para poder usar el entorno necesitaremos tener instalado, al menos, estas librerías de desarrollo con una versión igual

o superior a la 3.2. La versión del compilador *gcc* ha de ser también ésta, como mínimo. Cada uno de los componentes tiene dependencias de sus propias librerías y dependerá de la distribución de Linux que usemos, para saber el nombre del paquete que deberemos instalar antes de poder realizar la compilación.

1. 6 Familiarizarse con el IDE

Aunque un programa en Gambas se podría hacer perfectamente usando un editor de texto plano cualquiera, sería un desperdicio no aprovechar uno de los mayores atractivos que el lenguaje tiene: su IDE o entorno de desarrollo. El IDE de Gambas ahorra al programador buena parte del trabajo más tedioso, le proporciona herramientas que hacen mucho más fácil su tarea, con utilidades de ayuda, de diseño de interfaces, autocompletado de instrucciones, traducción de programas, etc. En la imagen siguiente podemos ver algunas de las ventanas más importantes del entorno, que se usan durante el desarrollo de una aplicación:

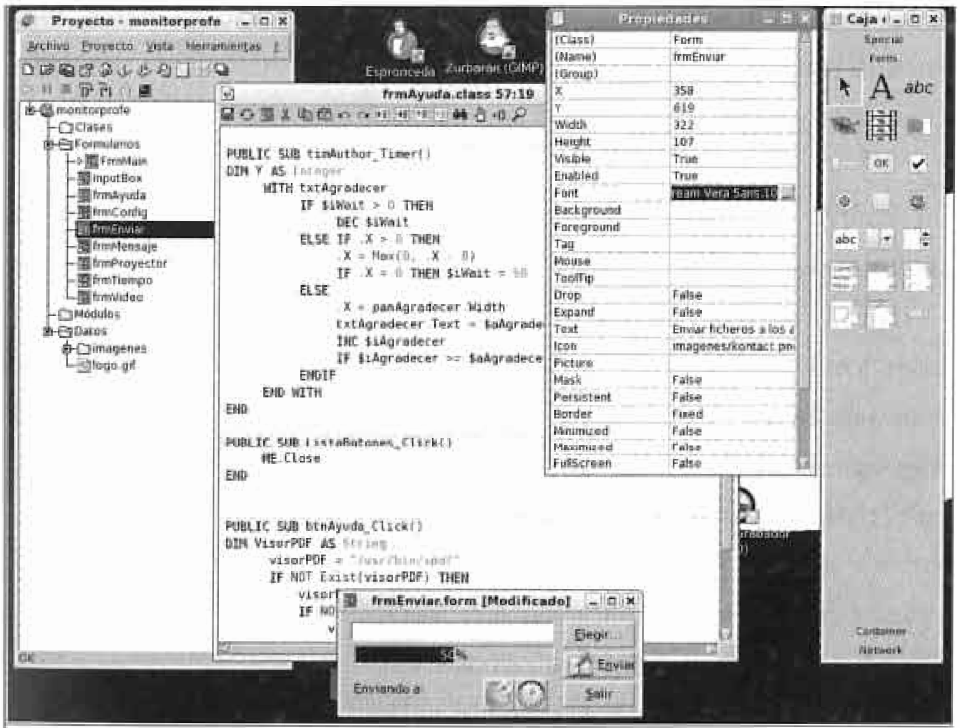


Figura 2. Entorno de desarrollo de Gambas.

Cuando se arranca Gambas, lo primero que nos aparece es la ventana de bienvenida.

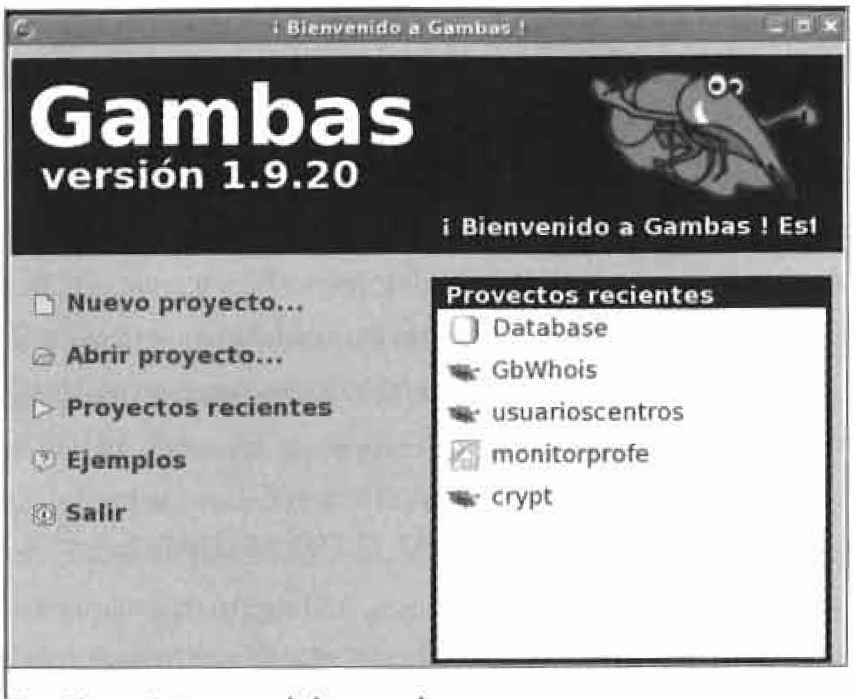


Figura 3. Ventana de bienvenida.

Aquí se nos ofrece la opción de comenzar un nuevo proyecto o aplicación, abrir un proyecto del que tengamos sus archivos disponibles, abrir uno usado recientemente o uno de los numerosos ejemplos que están incluidos en la ayuda de Gambas.

Antes de elegir cualquiera de estas opciones es necesario saber que todos los códigos fuente de una aplicación hecha en Gambas es lo que se denomina *proyecto*. El proyecto está formado por una serie de archivos que en Gambas están SIEMPRE situados dentro de un único directorio. En él puede haber, a gusto del desarrollador, distintos subdirectorios y organizar todo como se desee, pero cualquier gráfico, texto y código que forme parte de la aplicación estará dentro de él. Por ello, si elegimos en esta ventana la opción **Nuevo proyecto...**, el asistente siempre creará un nuevo directorio con el nombre del proyecto y ahí irá introduciendo todos los archivos necesarios para el desarrollo de la aplicación. Así, para enviar a alguien el código fuente de una aplicación hecha en Gambas o cambiarla de ordenador o disco, sólo hay que transportar el directorio con el nombre del proyecto, sin tener que preocuparse de otros archivos. Del mismo modo, si desde el entorno de desarrollo escogemos un

archivo o un gráfico para integrarlo en nuestro trabajo, el archivo será copiado automáticamente al directorio del proyecto.

□□□□□ El primer ejemplo

Una de las formas más habituales de empezar a trabajar con un lenguaje de programación es haciendo un pequeño programa que muestre el mensaje *Hola Mundo*. Por tanto, empezaremos a conocer el entorno de desarrollo y el lenguaje de programación con este típico ejemplo. Comenzaremos haciendo un *hola mundo* que sea puro BASIC, es decir, que sea igual al que hubieran hecho los autores de BASIC allá por el año 1964.

En aquellos tiempos las interfaces gráficas no existían, por lo que este primer programa será un programa feo, de terminal. En el BASIC original, hacer que aparezca un mensaje en el terminal es tan simple como escribir la línea:

```
PRINT "Hola Mundo"
```

No es necesario ningún encabezado previo, la instrucción **PRINT** sirve para mostrar cualquier cadena de texto en el terminal, que en BASIC se presentan entre comillas dobles. De ahí que el programa sea tan simple como ese. Vamos a ver cómo hacerlo con el entorno de desarrollo de GAMBAS:

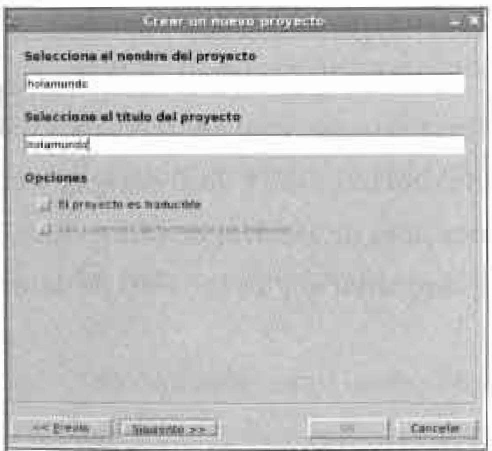


Figura 4. Asistente de Gambas.

1. Escogemos la opción **Nuevo proyecto...** en la ventana anterior (**Figura 3**). Aparecerá un asistente, en el que pulsamos **Siguiente**. Surgirá una nueva ventana para elegir el tipo de aplicación. Escogemos la tercera opción: **Crear un proyecto de texto**[7].

2. Le damos un nombre al proyecto, por ejemplo *holamundo* y un título. Pulsamos el botón **Siguiente** (**Figura 4**).

Elegimos el directorio del disco en el que queremos crearlo; pulsamos de nuevo el botón **Siguiente** y aparecerá un resumen con los datos del proyecto (**Figura 5**).



Figura 5. Datos del proyecto.

3. A continuación pulsamos el botón **OK**. Se abrirá el entorno de desarrollo de Gambas listo para empezar a programar. Al ser una aplicación de terminal, que no lleva interfaz gráfica, de momento, sólo necesitamos fijarnos en la ventana de la izquierda, que en nuestro caso tendrá como título: *Proyecto – holamundo* (**Figura 6**).

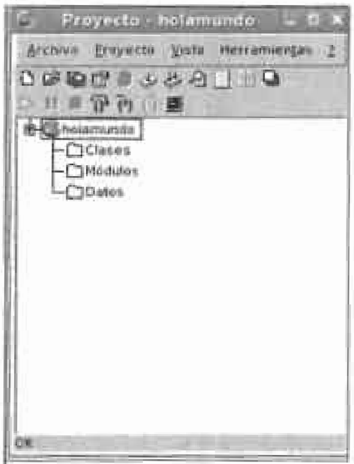


Figura 6. **Proyecto – holamundo**.

En realidad ésta es, probablemente, la ventana más importante para el manejo del entorno de desarrollo. A simple vista se puede ver el menú superior, que contiene las entradas necesarias para guardar y cargar proyectos, activar las distintas ventanas del IDE, manejar la ejecución de los programas, personalizar el entorno (en **Herramientas | Preferencias | Otros | Mostrar mascota**, podemos ocultar la animación de la gamba, que nos está mirando al abrir el proyecto), etc.

De momento nos fijamos sólo en el árbol de directorios que contiene. Podemos ver que la raíz del árbol es el nombre del proyecto: **holamundo**, y de él cuelgan tres ramas: **Clases** y **Módulos**, que son para distintos tipos de archivos de código fuente; y **Datos**, cuyo nombre indica su finalidad, almacenar ahí los archivos de datos que la aplicación requiera.

Desde el principio, Gambas nos da dos formas de realizar los programas, incluso si son tan simples como el que hemos hecho. Podemos elegir entre una programación

orientada a objetos, paradigma típico de los lenguajes de programación más potentes o una programación estructurada simple. Según ello, el archivo que contenga el código de nuestro programa será una *Clase* o un *Módulo*. Por simplicidad, de momento vamos a usar un *Módulo*. Haciendo clic con el botón derecho del ratón sobre el árbol de carpetas aparecerá un menú contextual. Elegimos las opciones **Nuevo | Módulo.** Surgirá una ventana en la que escribimos el nombre del módulo, por ejemplo *miprograma*, y pulsamos el botón **OK**, con lo que aparecerá nuestra primera ventana, con el título *miprograma.module*, donde poder escribir código.

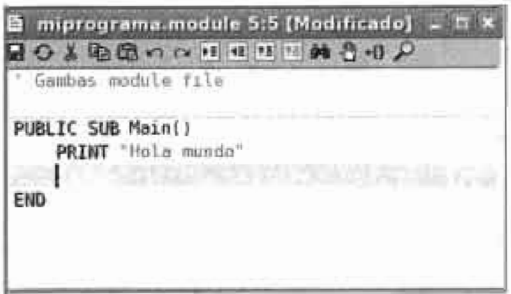


Figura 7. Ventana donde escribir el código.

Por fin podemos escribir nuestro código en BASIC. Lo haremos justo antes de la línea donde pone END, tal y como queda reflejado en la figura de la izquierda.

Cuando después de escribir el código que ya sabíamos, pulsamos la tecla INTRO, vemos que el entorno colorea el texto de una forma particular. Podemos pararnos un instante a ver los distintos colores que se muestran (**Figura 7**):

- En **gris** aparece una línea que comienza por una comilla simple ('). Esto indica que la línea es un comentario, es decir, no se trata de ningún código de programación y el texto que sigue a la comilla no se ejecuta nunca, son comentarios que el programador puede/debería poner para facilitar que otros (o él mismo, pasado un tiempo) entiendan lo que el programa hace en ese punto.

- En **azul** podemos ver palabras clave del lenguaje BASIC.

- En color **rosado** aparece la cadena de texto.

- A la izquierda vemos un resalte **amarillo** al comienzo de las líneas que han sido modificadas. Esto aparecerá siempre en las líneas que contengan modificaciones que no hayan sido compiladas.

Bien, ya está listo el programa. Para comprobarlo se pulsa en el botón verde, con el símbolo del **Play**, que está en la pantalla del proyecto. Al hacerlo aparecerá una nueva ventana llamada *Consola* en la que se verá la salida de nuestro programa. En este caso será el simple texto *Hola Mundo*. Éste es todo el código necesario, ya se puede compilar el programa para generar un archivo ejecutable que funciona sin necesidad del entorno de desarrollo. Para ello, en el menú de la ventana del proyecto hay que escoger: **Proyecto | Crear ejecutable**. Saldrá un cuadro de diálogo para elegir el directorio en el que queremos crear el ejecutable. Pulsando **OK** lo generará. Al cerrar ahora el entorno de desarrollo de Gambas y abrir un terminal o pasar a una consola de Linux, podemos probar su funcionamiento. Para ello, en el directorio donde se haya creado el ejecutable, hacemos:

```
jose@a00-o04:~$ cd gambas/holamundo/
jose@a00-o04:~/gambas/holamundo$ ./holamundo.gambas
Hola mundo
```


Mejor con un ejemplo gráfico

El ejemplo anterior mostraba una aplicación de consola, que nos recuerda a los viejos tiempos de otros sistemas operativos o a la forma de trabajar de los hackers informáticos. En realidad, hacer ese tipo de programas no demuestra el potencial de Gambas, puesto que son realmente simples e igualmente fáciles de realizar en otros lenguajes como Python o cualquier vieja versión de BASIC.

Es mejor hacer el programa *Hola Mundo* para el entorno gráfico que inunda los escritorios de los ordenadores actuales. Para ello empezaremos igual que antes, arrancando Gambas y creando un **Nuevo proyecto** siguiendo exactamente los mismos pasos, excepto que en lugar de escoger **Crear un proyecto de texto**, cuando el asistente nos presente las distintas opciones, elegiremos **Crear un proyecto gráfico.**

Para no repetir nombre, podemos denominar al proyecto *holamundo2*. Al acabar el proceso aparecerá de nuevo la ventana de proyecto, pero en esta ocasión tendrá una rama más en el árbol: **Formularios**. Un formulario es el área donde se diseña

la interfaz gráfica de la aplicación, es decir, donde se insertan objetos como botones, cuadros de texto, listas, casillas de verificación, etc. Los formularios se corresponderán con las ventanas que la aplicación mostrará.

Haciendo clic con el botón derecho del ratón sobre el árbol del proyecto, elegimos ahora en el menú contextual que aparece: **Nuevo | Formulario.** Por simplicidad, en este caso ni siquiera hace falta cambiar el nombre, sólo pulsando en el botón **OK** aparecerá en pantalla la ventana del formulario y una ventana para escribir código BASIC casi idéntica a la del ejemplo anterior, sin nada escrito. El resultado será algo parecido a lo siguiente:

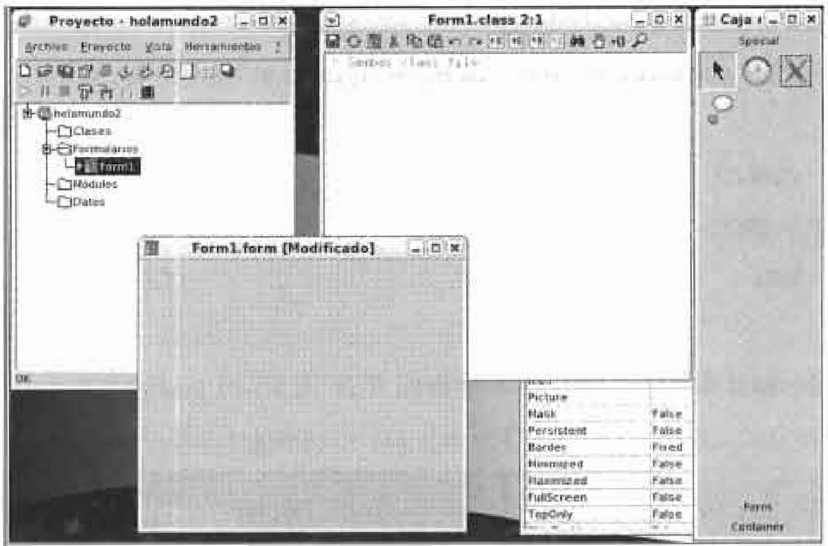


Figura 8. Pantalla del formulario.

Haciendo clic sobre la palabra **Form** en la ventana de la derecha, correspondiente a la **Caja de Herramientas**, aparecerán distintos objetos que podemos colocar en los formularios. Entre estos está el icono del botón (se distingue rápidamente por tener la palabra **OK** escrito dentro). Haciendo un clic de ratón en él, se puede dibujar en el formulario un botón, tan sólo hay que arrastrarlo con el ratón y el botón izquierdo pulsado, para dar la forma que queramos. Gambas escribe el texto *Button1* sobre el ratón, pero como ese texto no es demasiado intuitivo lo mejor es cambiarlo. Para ello hay que pinchar en el botón y después, en la ventana de propiedades, hacer un clic en la línea donde pone *Text*. Ahora se puede escribir el nuevo texto, por ejemplo **Púlsame.**

También, pulsando en la propiedad *Name* le cambiamos el nombre al objeto botón, poniéndole, por ejemplo, *btnMensaje*. Siguiendo los mismos pasos dibujamos otro botón sobre el formulario, escribimos el texto Salir y el nombre *btnSalir*. Una vez que tenemos diseñada la intefaz gráfica con el formulario y los dos botones, pasamos a escribir el código BASIC correspondiente. Para ello, haciendo un doble clic con el ratón sobre el botón con el texto *Púlsame*, aparecerá la ventana de código con un texto ya escrito, al que añadiremos algo de forma que el código de ese botón quede así:

```
PUBLIC SUB btnMensaje_Click()
  Message.Info("Hola Mundo")
END
```

Haciendo doble clic sobre el botón Salir, escribimos el código que falta:

```
PUBLIC SUB btnSalir_Click()
  QUIT
END
```

El resultado final debe ser algo como esto:

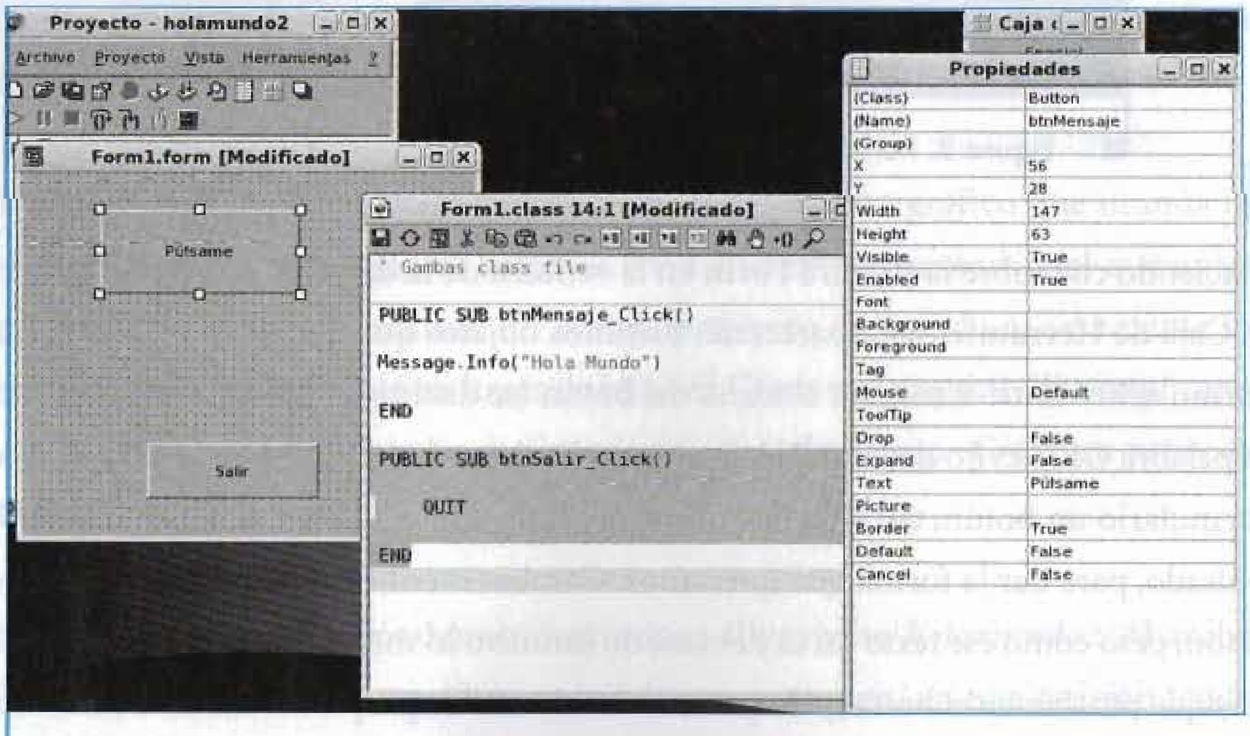


Figura 9. Resultado final de la programación de los dos botones.

Con esto ya está el programa terminado. Igual que antes, pulsando en el botón verde de la ventana de proyecto el programa se ejecutará. Sin embargo, en este caso la ejecución presenta novedades importantes: es un programa gráfico, tiene ya su ventana con la barra superior y los típicos botones de maximizar y minimizar. Además, cuenta con dos botones y la aplicación está esperando a que el usuario haga clic en alguno de ellos para actuar.

Hora de experimentar. Sería bueno, llegado este punto, que perdiéramos un poco de tiempo tocando las distintas propiedades de cada uno de los tres objetos que el programa tiene: el formulario y los dos botones, parando el programa y volviendo a arrancarlo para ver el resultado de las modificaciones tantas veces como queramos.

Aunque no es parte de este capítulo, hay un par de detalles que ya deberían empezar a observarse:

- El código BASIC se escribe entre unas líneas que empiezan por **PUBLIC SUB** y acaban en **END**.
- El texto que sigue a **PUBLIC SUB** lleva el nombre del objeto sobre el que se ha hecho un doble clic, seguido de la palabra **Click**, eso significa que el código se ejecutará cuando el usuario haga clic sobre el objeto.
- Para que aparezca **Message.Info**, se escribe **Message** y se pulsa sobre esta palabra. Aparece un menú entre cuyas opciones se encuentra la palabra **Info**. Moviéndonos con los cursores del teclado y pulsando la tecla de espacio o **INTRO** sobre esa palabra, quedaba escrito en la ventana de código. Ése es el autocompletado, que nos ayuda a escribir el código evitando tener que memorizar sentencias y mensajes extraños.

Un poco de magia

El entorno de desarrollo de Gambas está hecho con el componente *qt*, y de forma predeterminada lo escoge para las aplicaciones gráficas. Por tanto, un programa gráfico hecho en Gambas usa, en principio, estas librerías y se puede considerar

una aplicación nativa para el escritorio KDE. Sin embargo, el componente *gtk* disponible en la versión de desarrollo de Gambas, permite hacer programas enlazados con las librerías de *Gtk* para realizar aplicaciones del escritorio Gnome.

Si hemos instalado los paquetes de Gambas correspondientes a la rama de desarrollo, incluyendo el paquete *gambas-gb-gtk*, podemos recuperar la aplicación *holamundo2* del ejemplo anterior, ir a la ventana de *Proyecto* y hacer clic en el menú **Proyecto | Propiedades**, pestaña **Componentes**, y seleccionar **gb.gtk**. El resultado al ejecutar la aplicación es que se trata de una nueva, pero antes de KDE y ahora de Gnome. ¡Y sin tocar una sola línea de código!

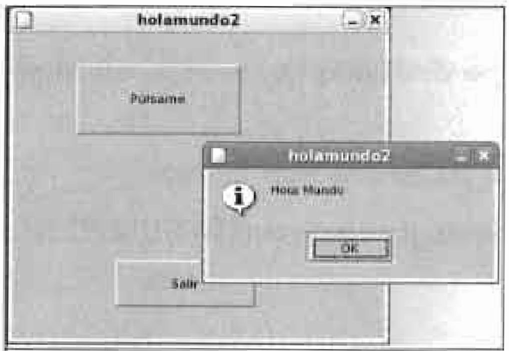


Figura 10. Aplicación *holamundo2* en KDE.



Figura 11. Aplicación *holamundo2* en Gnome.

Hasta el día de hoy no existe ningún otro lenguaje de programación con el que se pueda hacer esto.

1.7 Sistema de componentes

A lo largo de los apartados anteriores han aparecido distintas referencias a los componentes de Gambas, incluyendo su descripción en el tercer apartado de este capítulo. Estos permiten extender este lenguaje de programación. La interfaz desarrollada por Benoît para su programación hace que hayan sido varios los programadores que han colaborado con él, desarrollando nuevos componentes que se van añadiendo a las distintas versiones de Gambas en cada nueva publicación. En la versión 1.0 estable de

Gambas sólo se podían desarrollar en C y C++, pero desde la versión 1.9.4 de la rama de desarrollo también se pueden escribir componentes en Gambas, lo que abre numerosas posibilidades futuras ya que es mucho más sencillo hacerlo que en C.

El listado de componentes disponibles es amplio y se aumenta continuamente en la versión de desarrollo. Para la rama estable están fijados los siguientes:

- *gb.compress*: para compresión de archivos con protocolos *zip* y *bzip2*.
- *gb.qt*: para objetos visuales de las librerías gráficas *qt*.
- *gb.qt.ext*: para objetos visuales de las librerías gráficas *qt* que no son estándar.
- *gb.qt.editor*: un editor de texto hecho usando las librerías gráficas *qt*.
- *gb.qt.kde*: objetos visuales propios del escritorio KDE.
- *gb.qt.kde.html*: un navegador web del escritorio KDE.
- *gb.net*: objetos para conectar a servidores de red y a puertos serie de comunicaciones.
- *gb.net.curl*: objetos para construir servidores de red.
- *gb.db*: objetos de conexión a bases de datos.
- *gb.db.mysql*: driver para conectar al servidor de bases de datos MySQL.
- *gb.db.postgresql*: driver para conectar al servidor de bases de datos PostgreSQL.
- *gb.db.sqlite*: driver para usar bases de datos Sqlite 2.x.
- *gb.xml*: objetos para parsear archivos XML.

• *gb.vb*: colección de funciones para facilitar la migración desde Visual Basic.

• *gb.sdl*: objetos para reproducir, mezclar y grabar archivos de sonido.

• *gb.pcre*: objetos para usar expresiones regulares en el lenguaje.

En la rama de desarrollo existen estos componentes (algunos de los cuales han sido muy mejorados y aumentados, como el *gb.sdl*) y otros más, con una lista en continuo crecimiento. En el momento de escribir estas líneas podemos contar con:

• *gb.crypt*: objetos para encriptación DES y MD5.

• *gb.form*: objetos gráficos para formularios independientes de las librerías gráficas usadas.

• *gb.gtk*: objetos gráficos para formularios de las librerías *gtk*. Tiene los mismos objetos que el componente *gb.qt*, pero enlazan con esta otra librería de desarrollo.

• *gb.db.odbc*: driver para conectar a bases de datos a través de ODBC.

• *gb.info*: objetos que proporcionan distinta información acerca de los componentes y el sistema donde la aplicación se ejecuta.

• *gb.opengl*: objetos para dibujos tridimensionales con aceleración OpenGL.

• *gb.xml.rpc*: objetos para el uso del protocolo *rpc-xml*.

• *gb.sdl.image*: objetos para dibujos en dos dimensiones con aceleración gráfica.

• *gb.v4l*: objetos para capturar imágenes de cámaras de vídeo en Linux.

• *gb.gtk.pdf*: se utiliza para renderizar documentos *pdf* desde aplicaciones hechas en Gambas.

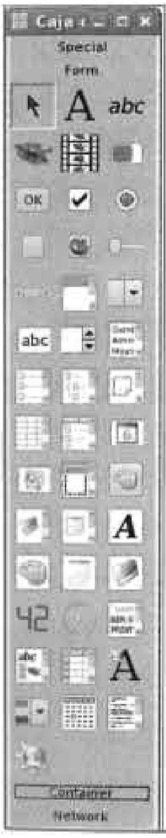
El listado de componentes disponibles para el programador puede verse en la pestaña **Componentes**, accesible a través del menú **Proyecto | Propiedades**. Cada uno de los componentes se corresponde a un paquete compilado en la distribución, de forma que si se añade al proyecto, por ejemplo, el componente *gb.sdl*, ha de instalarse el paquete *gambas-gb-sdl* en los ordenadores donde se quiera ejecutar la aplicación compilada.

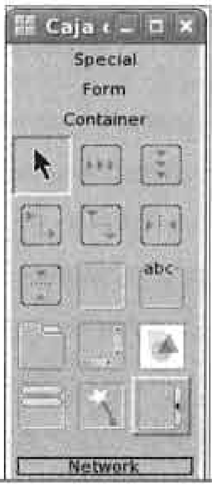


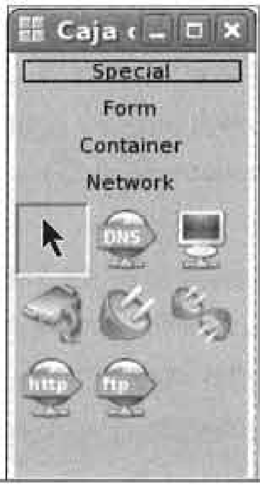
Figura 12. Componentes.

Al hacer clic sobre cada uno de los componentes aparece una pequeña descripción de su función, el nombre del autor o autores del componente, y un listado de los controles que están disponibles para el desarrollador si selecciona ese componente (**Figura 12**).

Los controles son clases para crear objetos útiles en la programación. Los objetos creados pueden ser visuales (como pestañas, editores de texto, etc.) u objetos de código (como servidores de red o conexiones a bases de datos). Si el componente tiene objetos visuales, estos se incorporan a alguna de las pestañas de la **Caja de Herramientas** del entorno de desarrollo.







Figuras 13, 14 y 15. **Caja de Herramientas** con componentes con objetos gráficos.

En las imágenes anteriores podemos ver algunos de los objetos gráficos que están disponibles al seleccionar los distintos componentes.

Cada componente tiene su propia documentación que se encuentra incluida en la ayuda de Gambas. En la rama estable esta documentación está siempre available; en la rama de desarrollo sólo está disponible si ha sido seleccionado para su uso en el proyecto.

Todas las cosas que se pueden hacer con Gambas, y no son parte del propio lenguaje BASIC, se programan mediante el uso de componentes. Esto significa que, por ejemplo, para hacer una aplicación de bases de datos es necesario seleccionar el componente *gb.db* o no estarán disponibles los objetos de conexión a bases de datos. Lo mismo ocurre con las conexiones de red, captura de vídeo, etc. Estos objetos no son parte del lenguaje BASIC.

NOTAS

1 **http://www.cooper.com/alan/father_of_vb.html**

2 Diferencias entre Gambas y Visual Basic: **http://wiki.gnulinex.org/gambas/210**

3 **http://www.fsf.org/licensing/licenses/gpl.html**. Hay una traducción (no oficial) al español en **http://gugs.sindominio.net/licencias/**

4 Documentación de Gambas en español: **http://wiki.gnulinex.org/gambas/**

5 Instrucciones para colaborar en la documentación en español: **http://wiki.gnulinex.org/gambas/6**

6 Si la instalación se hace desde un sistema gnuLinEx, no es necesario añadir esta línea puesto que la versión de desarrollo de Gambas es parte del repositorio oficial de gnuLinEx.

7 Debido a la mayor extensión de la cadena **Crear un proyecto de texto** en español que en otros idiomas, es posible que ese mensaje aparezca cortado y no se vea completo. No hay que preocuparse por ello, no afecta para nada a su elección y suponemos que será corregido en posteriores versiones del IDE.

2. Programación básica

2. Programación básica

2. Programación básica

2. Programación básica

● ○ En el *Capítulo 1* se vieron ya algunos detalles acerca de la programación con Gambas.

Ahora, estudiaremos, principalmente, la sintaxis e instrucciones más importantes del lenguaje BASIC que usa Gambas. Buena parte de estas instrucciones existen en el BASIC estándar y algunas otras son extensiones propias de Gambas.

Aunque sea más espartano, en este capítulo se usarán casi siempre ejemplos de código de consola, como vimos en el apartado *El primer ejemplo* del *Capítulo 1*. Haciéndolo de este modo evitaremos las interferencias que puede suponer la explicación de conceptos relacionados con la programación gráfica y los componentes. Estos conceptos se explicarán con extensión en capítulos posteriores.

2.1 Organización de un proyecto de Gambas

Antes de comenzar con la programación básica hay que resumir algunos conceptos previos:

- El código fuente de los programas hechos en Gambas está compuesto de uno o más archivos que forman un proyecto. Este proyecto se archiva en un directorio del mismo nombre.

- Los archivos de código pueden ser: *Módulos* (contienen código en BASIC que se ejecuta directamente), *Clases* (contienen el código en BASIC que ejecuta un objeto de esa clase) y *Formularios* (áreas donde se diseña la interfaz gráfica de la aplicación y que se corresponden con las ventanas del programa).

- Los proyectos de texto sólo contienen *Módulos* y/o *Clases*. Las aplicaciones gráficas contienen *Formularios* y *Clases*, pero también pueden contener *Módulos*.

- El proyecto puede contener otros archivos de datos, documentos, textos, etc., sin código BASIC para ser ejecutado por la aplicación.

Los archivos que contienen código en BASIC (*Módulos* y *Clases*) siempre están estructurados de la siguiente manera:

- Declaración de variables.

- Subrutinas y Funciones.

Declaración de variables

Los programas manejan datos continuamente. Estos datos pueden ser de muchos tipos: números, letras, textos, etc., y cambiar a lo largo de la ejecución del programa (en ese caso reciben el nombre de *variables*) o permanecer con un valor fijo durante todo el tiempo (entonces se denominan *constantes*). A los datos que usa un programa se les asigna un nombre identificador.

BASIC permitía usar variables y constantes sin haberlas declarado antes, es decir, sin haber expuesto al principio del programa un listado con las variables que se iban a usar.

Eso produce programas ilegibles en cuanto empiezan a ser de tamaño medio y es una fuente constante de errores. Para evitar esto, Gambas obliga a declarar las constantes y las variables antes de utilizarlas.

Hay dos lugares donde se pueden declarar las variables, dependiendo del ámbito en el que se vayan a usar. Si se declaran dentro de una subrutina o función (en el siguiente apartado se verá con detalle qué son las subrutinas y funciones), están disponibles para ser usadas sólo dentro de esa subrutina o función. Si se declaran al principio del archivo de código (sea un *Módulo* o una *Case*) están disponibles para todo el código de ese archivo, en todas sus subrutinas.

Para declarar una variable en una subrutina o función se emplea la sintaxis:

```
DIM nombre_variable AS tipo_variable
```


El **tipo_variable** hace referencia al tipo de dato de la variable: número entero, cadena de texto, número decimal, etc. El nombre de la variable lo elige el programador libremente. Siempre es recomendable que sea algo que indique para qué se va a usar la variable. Es decir, se debe huir de nombres como a, b, c, etc., y usar, por ejemplo, edad, fecha_nacimiento, altura, etc.

Las variables que se declaren en una subrutina o función sólo se usarán dentro de ellas. Cuando terminen se destruirán. Eso permite utilizar el mismo nombre de variable dentro de distintas subrutinas y su valor nunca se confundirá o mezclará.

Para declarar una variable al principio del *Módulo* o *Clase* usamos la sintaxis:

```
[STATIC] (PUBLIC | PRIVATE) nombre_variable AS
tipo_variable
```

Si se define la variable como **PRIVATE**, estará disponible dentro de todo el fichero, pero no será accesible desde otros ficheros del mismo proyecto. Si se la declara como **PUBLIC**, se podrá acceder a la variable desde un fichero del proyecto distinto a donde se declaró.

La palabra **STATIC** se usa en los archivos de *Clase*, no en los *Módulos*. Sirve para definir un comportamiento especial en todos los objetos de una misma clase. En pocas palabras se podría explicar con un ejemplo: si tenemos una clase perro, tendremos algunas variables como color, raza, peso, etc., y cada objeto perro tendrá su propio valor en cada una de esas variables.

Al mismo tiempo, podemos declarar una variable que sea *número_de_patas*, de forma que si cambiamos su valor de 4 a 3, todos los objetos perros tendrán 3 patas y cada uno seguirá con su propio peso, color, etc. En este caso, la variable *número_de_patas* se declararía *STATIC* en el código BASIC del archivo de la clase perro. Se verá todo este comportamiento más adelante en este mismo capítulo.

Las constantes se definen sólo al principio de un archivo de *Módulo* o *Clase*, no se pueden definir dentro de las subrutinas y funciones. La sintaxis es:

```
( PUBLIC | PRIVATE ) CONST nombre_constante AS
tipo_constante = valor
```

Por tanto, al igual que las variables, pueden ser privadas o públicas.

Veamos un ejemplo de todo esto:

```
' Gambas module file
' Archivo de Módulo con el nombre: ejemploVariables
PRIVATE edad AS Date
PRIVATE altura AS Single
PRIVATE CONST Pi = 3.141592
PUBLIC calidad AS Integer
```

```
PUBLIC SUB Subrutina1()
  DIM num AS Integer
  DIM nombre AS String
  edad=30
  num = 54
END

PUBLIC SUB subrutina2()
  DIM num AS Integer
  DIM apellido AS String
  edad=32
  num = 4
END
```

En este ejemplo vemos que las variables **edad** y **altura** se pueden usar en todo el archivo. Por tanto, después de ejecutar la **Subrutina1**, **edad** valdrá **30** y, después de ejecutar la **Subrutina2** valdrá 32. Vemos también cómo la variable **num** está definida en las dos subrutinas. El valor de **num** desaparece al acabar cada una de las subrutinas y, por tanto, durante **Subrutina1** valdrá **54** y durante **Subrutina2** valdrá **4**, pero después de que se ejecute el END de cada una de esas dos subrutinas, simplemente no existirá y si se hace referencia a *num* en cualquier otro punto del programa se producirá un error.

Finalmente, vemos que la variable **calidad** está definida como pública. Esto significa que desde cualquier otro archivo del programa se puede hacer referencia a ella mediante el nombre *ejemploVariables.calidad*, donde **ejemploVariables** es el nombre que se le ha dado al fichero donde se ha declarado **calidad**.

□□□□□ Subrutinas y funciones

Es impensable escribir todo el código de un programa sin una mínima organización. En BASIC el código se organiza dividiéndolo en procedimientos. Existen dos tipos de procedimientos: subrutinas y funciones. Una subrutina es un procedimiento que ejecuta algo, pero no devuelve ningún valor. Ejemplos de subrutinas

serían procedimientos para dibujar algo en la pantalla, tocar un sonido, etc. Sin embargo, una función es un procedimiento que devuelve siempre un valor al terminar su ejecución. Ejemplos de función serían el cálculo de una operación matemática que devuelve un resultado, el proceso para pedir datos al usuario de la aplicación, etc.

Ya hemos visto en el capítulo anterior la sintaxis para declarar las subrutinas, puesto que se mostró cómo el entorno de desarrollo escribe automáticamente la subrutina que el programa ejecuta al hacer un clic del ratón sobre un botón. La sintaxis completa es:

```
(PUBLIC | PRIVATE) SUB nombre_de_la_subrutina (p1 As
Tipo_de_Variable,p2 As Tipo_de_Variable, ...)
  ... código que ejecuta la subrutina
END
```


Las palabras PUBLIC y PRIVATE significan exactamente lo mismo que cuando se definen variables: determinan si la subrutina puede ser llamada sólo desde el archivo donde se ha codificado (*PRIVATE*) o desde cualquier archivo de la misma aplicación (*PUBLIC*).

Las variables **p1**, **p2**, etc., permiten pasarle parámetros a la subrutina, que se comportarán dentro de ella como variables declaradas dentro de la propia subrutina. Es decir, desaparecerán al ejecutar el END final. Se pueden pasar tantos parámetros como se desee a una subrutina, declarándolos todos, por supuesto.

Existen algunas subrutinas con nombres especiales en Gambas, por lo que el programador no debe usar esos nombres. Éstas son las siguientes:

- *Main*: existe en todas las aplicaciones de Gambas que sean de texto, no en las gráficas. Es el punto por el que empieza a ejecutarse el programa. Si no hubiera subrutina *Main*, Gambas daría un mensaje de error al intentar arrancar, puesto que no sabría por dónde comenzar.

• *_New* y *_free*: se ejecutan, respectivamente, al crearse y destruirse un objeto. Sólo se encuentran en los archivos de clase.

• *Objeto_Evento*: se ejecutan automáticamente cuando al Objeto le ocurre el Evento. Ya hemos visto algún ejemplo en el capítulo anterior, como *btnSalir_Click()*, que se ejecuta cuando el usuario de la aplicación hace clic con el ratón sobre el botón *btnSalir*. En las aplicaciones gráficas, el evento *Open* del formulario con el que se inicia la aplicación es la primera subrutina que el programa ejecutará. En el último apartado de este capítulo se tratarán específicamente los eventos, su significado y utilidad.

Veamos un programa de ejemplo (para probarlo, debemos crear un nuevo programa de texto siguiendo los pasos explicados en el *Capítulo 1*, apartado *El primer ejemplo*:

```
PUBLIC SUB Main()
  pinta_media(4,8)
END
PUBLIC SUB pinta_media(valor1 AS Integer, valor2 as
Integer)
  PRINT (valor1 + valor2)/2
END
```


Aunque éste es un programa poco útil, sirve para expresar con simplicidad la forma de funcionar de las subrutinas. Comienza ejecutando la subrutina **Main**, en ella sólo hay una llamada a ejecutar la subrutina **pinta_media** pasándole los números enteros **4** y **8** como parámetro. La subrutina **pinta_media** muestra en la consola el resultado de hacer la media entre los dos valores que han sido pasados como parámetros.

La sintaxis para declarar una función es la siguiente:

```
(PUBLIC | PRIVATE) FUNCTION nombre_de_la_función
(p1 As Tipo_de_Variable,p2 As Tipo_de_Variable ...)
As  Tipo_de_Dato
```

```
    ... código que ejecuta la función
    RETURN resultado_de_ejecutar la función
END
```

La declaración es casi idéntica a la de la subrutina, añadiendo dos cosas más: el tipo de dato que la función devuelve en la primera línea y la necesidad de usar la sentencia RETURN de BASIC para indicar el valor a devolver.

Veamos otro ejemplo que produce el mismo resultado que el anterior, pero usando una función:

```
PUBLIC SUB Main()
  DIM final AS Single
  final = calcula_Media(4,8)
  PRINT final
END

PUBLIC FUNCTION calcula_Suma33(valor1 AS Integer,
valor2 as Integer) AS Single
  RETURN (valor1 + valor2)/2
END
```


En esta ocasión es la subrutina **Main** la que se encarga de pintar en la consola el resultado de la operación. Es muy importante destacar la diferencia entre la forma en que se llama a una subrutina y se llama a una función. En el ejemplo anterior vimos que para llamar a la subrutina sólo se escribía su nombre con sus parámetros entre paréntesis. Sin embargo, ahora vemos que al llamar a la función se usa una asignación, **final** = **calcula_Media(4,8)**. Esto debe hacerse siempre al llamar a una función: la asignación sirve para recoger el valor que se devuelve. Por motivos obvios, la variable que recoge el valor que la función retorna debe declararse del mismo tipo de dato que el que devuelve la función. En el ejemplo anterior la función devuelve un dato tipo **Single** (un número real con decimales) y la variable **final** se ha declarado, por tanto, de tipo **Single**.

2. 2 Tipos de datos

Ya hemos visto a lo largo de los textos anteriores el uso de variables y cómo se declaran.

Hasta el momento, sólo conocemos algunos tipos de datos. Revisemos ahora todos los que soporta Gambas:

- *Boolean*: es un tipo de dato que suele ser el resultado de una comparación. Sólo acepta dos valores: *False* y *True* (Falso y Verdadero en español).

- *Byte*: representa un número entero positivo entre 0 y 255.

- *Short*: representa un número entero con valores posibles entre -32.768 y +32.767.

- *Integer*: representa un número entero con valores posibles entre -2.147.483.648 y +2.147.483.647.

- *Long*: representa un número entero con valores posibles entre -9.223.372.036.854.775.808 y +9.223.372.036.854.775.807.

- *Single*: representa un número real, con decimales, con valores posibles entre -1,7014118+38 y +1,7014118E+38.

- *Float*: representa un número real, con decimales, con valores posibles entre -8,98846567431105E+307 y +8,98846567431105E+307.

- *Date*: sirve para almacenar un valor de fecha y hora. Internamente, la fecha y hora se almacena en formato UTC, al devolver el dato se representa en el formato local, según la configuración del ordenador.

- *String*: se usa para almacenar una cadena de texto. En BASIC las cadenas de texto se asignan mediante dobles comillas.

- *Variant*: significa *cualquier tipo de dato*, es decir, puede almacenar un *Integer*, *Single*, *String*, etc. Se debe evitar su uso porque ocupa más memoria que los anteriores y los cálculos con *Variant* son mucho más lentos.

- *Object*: representa cualquier objeto creado en Gambas.

Será el programador el que elija el tipo de dato con el que debe ser declarada una variable. Siempre se ha de tender a usar los tipos de datos más pequeños, puesto que ocupan menos memoria y el microprocesador los maneja con más velocidad. Sin embargo, eso puede limitar las opciones de la aplicación, por lo que a menudo se opta por tipos mayores para no cerrar posibilidades.

Por ejemplo, si se va a usar una variable para definir la edad de una persona, lo lógico es utilizar un dato de tipo byte (el valor máximo es 255). Sin embargo, si la edad a guardar es la de un árbol, es conveniente usar un tipo *Short*, ya que puede haber árboles con más de 255 años, pero no se conocen con más de 32.767.

Conversión de tipos

Gambas permite distintas conversiones entre tipos de datos. La forma de hacer la conversión puede ser implícita o explícita. Son conversiones implícitas cuando el propio intérprete de Gambas se encarga de gestionarlas. Por ejemplo:

```
DIM resultado as Single
DIM operando1 as Single
DIM operando2 as Integer
operando1= 3.5
operando2=2
resultado = operando1 * operando2
```

En este caso, para poder realizar la multiplicación, el intérprete convierte, de forma transparente para el programador, el **operando2** a un valor **Single**. Son conversiones explícitas las que debe hacer el programador al escribir el código para poder realizar operaciones, mezclar datos de distintos tipos, etc.

Estas conversiones se hacen mediante unas funciones que están incluidas en el BASIC de Gambas. Evidentemente, la conversión se hará siempre que sea posible, si no lo es, producirá un error. Éste es el listado de funciones de conversión existente:

- *CBool(expresión)*: convierte la expresión a un valor booleano, verdadero o falso. El resultado será falso si la expresión es falsa, el número 0, una cadena de texto vacía o un valor nulo. Será verdadero en los demás casos. Por ejemplo:

  - Devuelven *False* las siguientes operaciones: *Cbool("")*, *Cbool(0)*, *Cbool(NULL)*.

  - Devuelven *True* las operaciones: *Cbool(1)*, *Cbool("Gambas")*.

- *CByte(expresión)*: convierte la expresión en un valor tipo *Byte*. Primero se convierte la expresión a un número binario de 4 bytes. Si este número es mayor de 255, se corta recogiendo los 8 bits de menor peso. Por ejemplo, *Cbyte("17")* devuelve 17, pero *Cbyte*(100000) devuelve 160. El valor 160 es debido a que el número 100.000 en binario es 11000011010100000, sus 8 últimos bits son 10100000, que pasado de binario a decimal da lugar a 160. Si no sabemos operar con números binarios lo mejor que se puede hacer es evitar este tipo de conversiones que resultan en valores tan "sorprendentes".

- *CShort(expresión)*, *CInt(expresión)* o *CInteger(expresión)*, y *CLong(expresión)*: convierten, respectivamente, la expresión en un número de tipo *Short*, *Integer* y *Long*. En el caso de *CShort* la conversión se realiza igual que para *CByte*, pudiendo producirse resultados extraños igualmente si la expresión resulta en un número mayor de 32.767.

- *CDate(expresión)*: convierte la expresión en una fecha, pero hay que tener cuidado porque no admite el formato de fecha local, sólo el formato inglés *mes/día/año horas:minutos:segundos*. Es decir, *CDate("09/06/1972 01:45:12")* es el día 6 de septiembre de 1972.

- *CSingle(expresión)* o *CSng(expresión)* y *CFoat(expresión)* o *Cflt(expresión)*: convierten, respectivamente, la expresión en un número del tipo *Single* y *Float*. La expresión debe usar el punto (.) y no la coma (,) como separador decimal.

- *CStr(expresión)*: convierte la expresión en una cadena de texto sin tener en cuenta la configuración local. Por tanto, *Cstr(1.58)* devuelve la cadena de texto *1.58*, independientemente de si la configuración local indica que el separador decimal es el punto o la coma, o *CStr(CDate("09/06/1972 01:45:12"))* devuelve *"09/06/1972 01:45:12"*.

- *Str$(expresión)*: convierte la expresión en una cadena de texto, teniendo en cuenta la configuración local. Por tanto, *Str$(1.58)* devuelve la cadena de texto *1,58*, si la configuración local está en español. Del mismo modo, *Str$(CDate("09/06/1972 01:45:12"))* devuelve *"06/09/1972 01:45:12"* si la configuración local está en español, puesto que en este idioma la costumbre es escribir las fechas en la forma día/mes/año.

- *Val(expresión)*: convierte una cadena de texto en un tipo *Boolean*, *Date* o alguno de los tipos numéricos, dependiendo del contenido de la expresión. *Val* es la función opuesta de *Str$* y, por tanto, también tiene en cuenta la configuración local del ordenador en el que se ejecuta. Intenta convertir la expresión en un tipo *Date*, si no puede en un número con decimales, si tampoco puede en un número entero y si, finalmente, tampoco puede la convierte en un tipo *Boolean*.

Matrices

En numerosas ocasiones, cuando se intenta resolver un problema mediante la programación, surge la necesidad de contar con la posibilidad de almacenar distintos datos en una misma variable. La solución más simple para este problema son las *Matrices* o *Arrays*. Se pueden definir matrices que contengan cualquier tipo de datos, pero con la condición de que todos los elementos de la matriz sean del mismo tipo. No hay más límite en la dimensión de la matriz que la memoria del ordenador y la capacidad del programador de operar con matrices de dimensiones grandes.

La sintaxis para trabajar con matrices es la misma que para las variables, añadiendo entre corchetes las dimensiones de la matriz. Algunos ejemplos:

```
DIM Notas[2, 3] AS Single
DIM Edades[40] AS Integer
PRIVATE Esto_no_hay_quien_lo_maneje[4, 3, 2, 5, 6, 2]
AS String
```

Para acceder al valor de cada una de las celdas de la matriz, hay que referirse siempre a su índice. En una matriz de dos dimensiones lo podemos identificar fácilmente por filas y columnas.

Hay que tener en cuenta que el índice comienza a contar en el 0, no en el 1. Es decir, en la matriz *Listado[3]* existirán los valores correspondientes a *Listado[0]*, *Listado[1]* y *Listado[2]*. Si se intenta acceder a *Listado[3]* dará un error *Out of Bounds*, fuera del límite. Por ejemplo:

```
Dim Alumnos[4,10] AS String
Dim columna AS Integer
Dim fila AS Integer
Columna= 2
Fila= 6
Alumno[Columna, Fila]="Manolo Pérez Rodríguez"
```

Hemos visto que hay que declarar estas matrices con las dimensiones máximas que van a tener. Eso supone que el intérprete de Gambas reserva la memoria necesaria para ellas al comenzar el uso del programa. Sin embargo, hay veces en que, por las características de la aplicación, se desconoce la dimensión que hará falta para la matriz. Para solventar esta posibilidad Gambas tiene predefinidas matrices unidimensionales dinámicas de todos los tipos de datos, excepto *Boolean*. Con estas matrices se trabaja de forma distinta a las anteriores, ya que hacen falta funciones para añadir nuevos elementos a la matriz, borrarlos o saber el número de elementos que tiene ésta.

Con este ejemplo veremos el uso y funciones más usuales al trabajar con matrices dinámicas:

```
DIM nombres AS String[]
' La siguiente instrucción inicializa nombres para
poder usarlo.
' Es un paso previo obligado:
nombres = NEW String[]
' Así podemos añadir valores a la matriz:
nombres.Add("Manolo")
nombres.Add("Juan")
nombres.Add("Antonio")
'Count devuelve el número de elementos de la matriz.
'La siguiente instrucción pintará 3 en la consola
PRINT nombres.Count
'La siguiente instrucción borrará la fila de "Juan":
nombres.Remove(1)
PRINT nombres.Count 'pintará 2
PRINT nombres[1] 'pintará "Antonio"
'La siguiente instrucción vaciará nombres:
nombres.Clear
PRINT nombres.Count 'pintará 0
```


2. 3 Operaciones matemáticas

Cuando se trata de trabajar con números, Gambas tiene las operaciones habituales en casi todos los lenguajes de programación:

- +, - , * y / se usan, respectivamente, para la suma, resta, multiplicación y división.

- ^ es el operador de potencia, por ejemplo, *4 ^ 3 = 64*.

- Para la división hay dos operadores adicionales, \ o *DIV* y *MOD*, que devuelven, respectivamente, la parte entera del resultado de la división y el resto. Es decir, *9 DIV 2 = 4, 9 \ 2= 4* y *9 MOD 4=1P.*

Aparte de estos operadores existen las siguientes funciones matemáticas para realizar cálculos más complejos:

- *Abs(número)*: devuelve el valor absoluto de un número.

- *Dec(número)*: decrementa un número.

- *Frac(número)*: devuelve la parte decimal de un número.

- *Inc(número)*: incrementa un número.

- *Int(número)*: devuelve la parte entera de un número.

- *Max(número1, número2, ...)*: devuelve el número mayor.

- *Min(número1, número2, ...)*: devuelve el número menor.

- *Round(número,decimales)*: redondea un número con los decimales deseados.

- *Sgn(número)*: devuelve el signo de un número.

- *Rnd([mínimo],[máximo])*: devuelve un número aleatorio comprendido entre *mínimo* y *máximo*. Si no se expresa ningún valor para mínimo y máximo, el número estará comprendido entre 0 y 1. Si sólo se expresa un valor, el número estará comprendido entre el 0 y ese valor. Muy importante: antes de usar *Rnd* es necesario ejecutar la instrucción *Randomize* que inicializa el generador de números aleatorios. Si no se hace, se obtendrá el mismo número en sucesivas ejecuciones de *Rnd*.

Operaciones lógicas

Para realizar operaciones entre variables de tipo *Boolean* o expresiones cuyo resultado sea *Boolean*, existen algunas instrucciones similares a las que podemos ver en casi todos los lenguajes de programación. Se trata *AND*, *OR*, *NOT* y *XOR*. Si tenemos conocimientos de lógica y números binarios, no nos resultará difícil identificarlas y saber su comportamiento al tratarse de las operaciones binarias más básicas. En caso contrario, será fácil usarlas y entender su funcionamiento con una simple traducción al español de las tres primeras: *Y*, *O*, *NO*. Es decir, sirven para unir condiciones del tipo: color es verde y no es marrón, que se escribiría:

```
Color="Verde" AND NOT Color="Marrón"
```

El caso de *XOR* es más difícil de entender puesto que es una operación algo especial llamada *OR exclusivo*. El resultado de una operación *XOR* es verdadero cuando los dos operandos son distintos y, falso, cuando los dos operandos son iguales. En la práctica, esta operación se utiliza en cálculos con números binarios, en cuyo caso seguro que conocemos perfectamente su funcionamiento.

2. 4 Manejo de cadenas

Una de las tareas más habituales en los programas informáticos es el uso de cadenas de texto, tanto si se trata de aplicaciones de bases de datos, como para la simple salida de mensajes en pantalla. En Gambas se han implementado todas las funciones de cadenas de texto del BASIC estándar más las que están presentes en Visual Basic. Antes de proceder a su listado, destacar que existe un "operador" de cadenas de texto que permite concatenarlas directamente, se trata del símbolo &. Veamos un ejemplo de su uso:

```
Dim Nombre AS String
Dim Apellidos AS String
Nombre = "Manuel"
Apellidos = "Álvarez Gómez"
PRINT Apellidos & ", " & Nombre
```

La salida en consola será:

```
Álvarez Gómez, Manuel
```

Veamos ahora el listado de las funciones disponibles para manejar cadenas de texto:

- *Asc(Cadena,[Posición])*: devuelve el código ASCII[1] del carácter que está en la posición indicada en la cadena dada. Si no se da la posición, devuelve el código del primer carácter.

- *Chr$*: devuelve un carácter a partir de su código ASCII. Esta función es útil para añadir caracteres especiales a una cadena de texto, por ejemplo:

```
PRINT "Manuel" & Chr$(10) & "Antonio"
```

  insertará una tabulación entre los dos nombres, ya que en la tabla ASCII el código **10** corresponde a un avance de línea (*Line Feed*).

- *InStr (Cadena, Subcadena [, Inicio])*: busca la subcadena dentro de la cadena y devuelve un número con la posición donde la encontró. Si se da el valor *Inicio*, la búsqueda empezará en esa posición. Por ejemplo:

```
PRINT Instr("Gambas es basic", "bas", 5)
```

  devuelve un *11*, mientras que:

```
PRINT Instr("Gambas es basic", "bas")
```

  devuelve un *4*.

- *RinStr (Cadena, Subcadena [, Inicio])*: funciona igual que *InStr*, sólo que empieza a buscar de derecha a izquierda en la cadena.

- *Lcase$(Cadena)*: convierte una cadena a minúsculas.

- *Ucase$(cadena)*: convierte una cadena a mayúsculas.

- *Left$(cadena,n)*: devuelve los primeros *n* caracteres de una cadena.

- *Right$(cadena,n)*: devuelve los últimos *n* caracteres de una cadena.

- *Mid$(Cadena, posición [,n])*: devuelve una cadena que empieza en *posición* y mide *n* caracteres. Si no se especifica *n*, la cadena devuelta contendrá todos los caracteres a partir de *posición*.

- *Replace$(Cadena, patrón, reemplazo)*: reemplaza, dentro de *Cadena*, cada vez que encuentra el texto *patrón* por el texto *reemplazo*. Por ejemplo:

```
PRINT Replace$("LinEx es una distribución de Linux",
"Lin", "gnu/Lin")
```

pinta el texto:

```
"gnu/LinEx es una distribución de gnu/Linux"
```

- *Subst$(Patrón, reemplazo1[, reemplazo2,...])*: sustituye en la cadena *patrón* los símbolos *&1*, *&2*... por las cadenas *reemplazo1*, *reemplazo2*... Por ejemplo:

```
PRINT Subst$("El alumno &1 tiene &2 años","Manuel","8")
```

da como salida la cadena:

```
"El alumno Manuel tiene 8 años".
```

Evidentemente, el potencial de esta instrucción se muestra cuando las cadenas de reemplazo son variables de tipo *String*.

- *Len(cadena)*: devuelve la longitud de una cadena.

- *Ltrim$(cadena)*: elimina los espacios en blanco de la parte inicial de una cadena.

- *Rtrim$(cadena)*: elimina los espacios en blanco de la parte final de una cadena.

- *Trim$(cadena)*: elimina todos los espacios en blanco de una cadena.

- *Space$(n)*: devuelve una cadena que contiene *n* espacios en blanco.

- *String$(n, patrón)*: devuelve una cadena que contiene el *patrón* concatenado *n* veces.

- *Split(Cadena [, Separadores , Escape , Ignorar_nulos])*: devuelve una matriz de cadenas de texto. Las partes de esta matriz se obtienen al trocear la *Cadena* cada vez que se encuentra uno de los *Separadores*. Si no se especifica ninguno, el separador se considera la coma (,). Si en algún caso se quiere incluir el *Separador* en alguna de las cadenas separadas, habría que colocar el *Escape* marcando el principio y final de la parte en la que debe ignorarse el separador.

- *Ignorar_nulos* es un valor *True* o *False*, e indica si se deben ignorar o no las cadenas que resulten vacías como consecuencia del troceado. Veamos un ejemplo que explique mejor el funcionamiento:

```
DIM texto as String[]
DIM texto2 as String[]
texto = Split("Había una vez un circo", " ")
texto2 = Split("Había una vez ¡un circo¡", " ", "¡")
PRINT texto[0] & "," & texto[1] & "," & texto[2] & ","
& texto[3]
PRINT texto2[0] & "," & texto2[1] & "," & texto2[2]
& "," & texto2[3]
```

devuelve:

```
Había,una,vez,un
Había,una,vez,un circo
```

En el segundo caso se puede ver cómo, aunque el separador es el espacio en blanco, no se ha separado el texto **un circo** al estar rodeado del carácter de escape.

2. 5 Control de flujo

Son muchas las ocasiones en que el flujo en que se ejecutan las instrucciones en un programa no es adecuado para resolver problemas.

Todo el código BASIC que hemos visto hasta ahora ejecuta sus instrucciones de arriba abajo, según se las va encontrando. Sin embargo, con frecuencia, habrá que volver atrás, repetir cosas, tomar decisiones, etc. Este tipo de acciones se denomina *control de flujo* y el BASIC de Gambas proporciona un buen juego de sentencias para manejarlo.

IF ... THEN ... ELSE

Es la sentencia más común para tomar una decisión: si se cumple una condición, entonces se ejecuta algo, en caso contrario, se ejecuta otra cosa.

Su forma más básica es:

```
IF Expresión  THEN
   ...
ENDIF
```

O si lo que se ejecuta es una sola instrucción:

```
IF Expresión  THEN sentencia_a_ejecutar
```

La sintaxis completa de la instrucción es:

```
IF Expresión [ { AND IF | OR IF } Expresión ... ] THEN
  ...
[ ELSE IF Expresión [ { AND IF | OR IF } Expresión ... ]
THEN
  ... ]
[ ELSE
  ... ]
ENDIF
```

Algunos ejemplos de uso:

```
DIM Edad AS Integer

.....

IF Edad > 20 THEN
  PRINT "Adulto"
ENDIF
IF Edad > 20 THEN PRINT "Adulto"
IF Edad < 2 AND Edad >0 THEN
  PRINT "Bebé"
ELSE IF Edad < 12 THEN
  PRINT "Niño"
ELSE IF Edad < 18 THEN
  PRINT "Joven"
ELSE
  PRINT "Adulto"
ENDIF
```

Dependiendo del valor que tuviera la variable **Edad** al llegar al primer **IF**, se imprimirá un resultado distinto.

□□□□□ Select

En el caso anterior vimos que en ocasiones el flujo del programa necesita revisar varias condiciones sobre una misma variable, produciendo un *IF* dentro de otro (*IF* anidados). Esa estructura no es cómoda de leer ni produce un código limpio. Para estos casos existe la sentencia SELECT, que es mucho más apropiada. Su sintaxis es:

```
SELECT [ CASE ] Expresión
[ CASE Expresión [ TO Expresión #2 ] [ , ... ]
... ]
[ CASE Expresión [ TO Expresión #2 ] [ , ... ]
... ]
[ { CASE ELSE | DEFAULT }
... ]
END SELECT
```

Veamos cómo se aplica al mismo ejemplo anterior de las edades:


```
DIM Edad AS Integer
....
SELECT CASE edad
CASE 0 TO 2
  PRINT "Bebé"
CASE 2 TO 12
  PRINT "Niño"
CASE 18
  PRINT "Bingo, ya puedes votar"
CASE 13 TO 17
  PRINT "Joven"
CASE ELSE
  PRINT "Adulto"
END SELECT
```

Se trata de un código mucho más fácil de leer que el anterior.

□□□□□ FOR

Cuando se hace necesario contar o realizar una acción un número determinado de veces, la sentencia FOR es la solución:

```
FOR Variable = Expresión TO Expresión [ STEP Expresión ]
  ...
NEXT
```

El bucle incrementa la **Variable** de uno en uno, a no ser que se especifique un valor a STEP. Se pueden especificar valores negativos, de forma que se convertiría en una cuenta atrás. Por ejemplo:

```
DIM n AS Integer
FOR n = 10 TO 1 STEP -1
  PRINT n
NEXT
```


Si se quiere interrumpir un bucle en algún punto, se puede usar la sentencia BREAK:

```
DIM n AS Integer
FOR n = 10 TO 1 STEP -1
  IF n=3 THEN BREAK
  PRINT n
NEXT
```

El bucle acabaría cuando **n** valiera 3 y no se escribirían los últimos 3 números. También se dispone de la sentencia CONTINUE, que permite saltarse pasos en el bucle:

```
DIM n AS Integer
FOR n = 1 TO 4
  IF n=2 THEN CONTINUE
  PRINT n
NEXT
```

Se saltaría el 2 al escribir los valores de **n**. Existe una variante del bucle **FOR** que se usa al recorrer elementos de una colección, como una matriz. La sintaxis en este caso es:

```
FOR EACH Variable IN Expresión
...
NEXT
```

Pongamos un ejemplo usando las matrices dinámicas que ya hemos visto en este capítulo:

```
DIM Matriz AS String[]
DIM Elemento AS String

Matriz = NEW String[]
Matriz.Add("Azul")
Matriz.Add("Rojo")
Matriz.Add("Verde")

FOR EACH Elemento IN Matriz
  PRINT Elemento;
NEXT
```


Escribiría como salida: *AzulRojoVerde*.

□□□□□ WHILE y REPEAT

Cuando se quiere repetir la ejecución de una porción de código en varias ocasiones dependiendo de una condición, tenemos dos instrucciones distintas: *WHILE* y *REPEAT*.

Su comportamiento es casi idéntico. La diferencia estriba en que si la condición necesaria para que se ejecute el código es falsa desde el principio, con **REPEAT** se ejecutará una vez y con **WHILE** no se ejecutará nunca. La sintaxis de ambas es:

```
WHILE Condición
  ... instrucciones
WEND
```

y

```
REPEAT
  ...instrucciones
UNTIL Condición
```

En el caso del bucle WHILE existe una variante de la sintaxis consistente en sustituir WHILE por *DO WHILE* y WEND por *LOOP*. Es exactamente lo mismo; depende del programador elegir un formato u otro. Veamos un ejemplo:

```
DIM a AS Integer
a = 1
WHILE a <= 10
  PRINT "Hola Mundo "; a
  INC a
WEND
 a = 1

REPEAT
  PRINT "Hola Mundo "; a
  INC a
UNTIL a > 10
```


Este ejemplo producirá el mismo resultado en la ejecución del bucle WHILE que en el del REPEAT, en ambos casos escribirá diez veces "**Hola Mundo** " junto al valor de la variable a que se irá incrementando de 1 a 10. El uso de estas estructuras puede ser peligroso. Si durante la ejecución del bucle no hay forma de que la condición pase de ser verdadera a falsa, estaríamos ante un bucle infinito y el programa entraría en situación de bloqueo.

Depuración en el IDE de Gambas

Una vez escrito parte del código de un programa, lo usual es comprobar si funciona, pero lo habitual es que la mayor parte de las veces no lo haga en el primer intento. Tanto si se es un programador experimentado como si no, los fallos son parte de la rutina. Saber encontrarlos y corregirlos es lo que se denomina depuración, y es una de las tareas más importantes a realizar. Cuando son fallos de sintaxis, el entorno de desarrollo suele darnos mensajes indicativos del problema, parando la ejecución en la línea en que se produce.

Cuando se adquiere una cierta soltura con el lenguaje, los fallos de sintaxis son cada vez menos comunes, pero los fallos en la lógica de la ejecución del programa se producirán siempre. Cuando esa lógica pasa, además, por instrucciones de control de flujo como las que hemos visto en este capítulo, la dificultad en encontrar los errores es mayor, puesto que la aplicación transcurre en distintas ocasiones por la misma porción de código y es posible que el fallo no se produzca la primera vez que se ejecute ese código.

Para facilitar esta tarea, el IDE de Gambas dispone de distintas herramientas de depuración. La primera de ellas es la posibilidad de fijar puntos de interrupción. Es decir, señalar sitios en los que el programa se parará para permitir ver el estado de las variables y en qué punto de la ejecución se encuentra.

Para fijar un punto de interrupción en una línea de código, tan sólo hay que colocar el cursor del ratón en esa línea y pulsar la tecla **F9** o el botón con el símbolo de la mano levantada, que está en la parte superior de la ventana de código. Las líneas en las que se fija un punto de interrupción tienen el fondo en rojo.

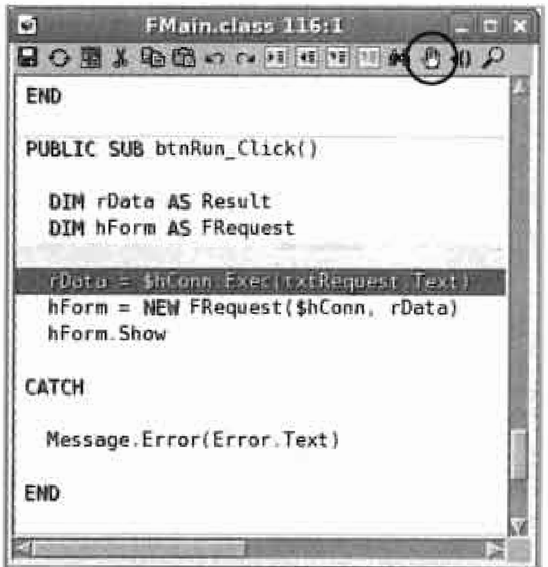


Figura 1. Punto de interrupción en la línea de código.

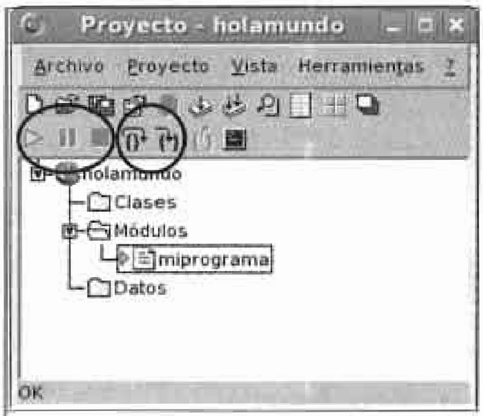


Figura 2. Ejecución del programa desde la ventana **Proyecto**.

La misma operación que crea el punto la elimina, es decir, pulsando **F9** de nuevo el fondo rojo desaparecerá. La ejecución del programa, como se explicó en el capítulo anterior, se arranca pulsando en el símbolo verde de **Play** de la ventana de **Proyecto** (o pulsando la tecla de función F5). Junto al botón verde se encuentra un botón de **Pausa**, que permite parar la ejecución, y otro más de **Stop** que permite detenerla en cualquier momento.

Si se quiere correr la aplicación ejecutando una a una las instrucciones para ir observando por dónde transcurre el flujo del programa, se puede pulsar la tecla de función **F8** o cualquiera de los dos botones que se encuentran a la derecha del símbolo de **Stop**. Haciendo esto, el entorno de desarrollo saltará a la primera línea que se deba correr e irá ejecutando línea a línea cada vez que pulsemos **F8**, o el icono que muestra la flecha entrando entre las llaves, mencionado anteriormente. El botón que está justo a la derecha del **Stop** y que muestra una flecha saltando por encima de las llaves, parece producir el mismo efecto (su tecla rápida es **Mayúsculas + F8**), pero no es así: el comportamiento cambia cuando el programa llegue a una llamada, a una subrutina o a una función. En este caso, este icono ejecutará la llamada y todo lo que tenga que hacer la función o subrutina en un solo paso, sin que veamos el flujo del programa por el procedimiento. Si hubiéramos pulsado **F8** habríamos entrado en la subrutina y visto también paso a paso cómo se ejecutan las instrucciones. Por tanto, con estos dos botones podemos elegir cuando lleguemos a la llamada a un procedimiento, si queremos depurar también ese procedimiento o simplemente ejecutarlo y pasar a la siguiente línea de código.

Finalmente, cuando pausamos la ejecución del programa, aparecen tres nuevas pestañas en la parte inferior de la ventana de proyecto. En la pestaña **Local** se ven todas las variables del procedimiento que se está ejecutando y el valor que tienen en ese momento. En la pestaña **Pila** se ve el listado de llamadas entre procedimientos que se han producido hasta llegar a ese punto del programa. Así, podemos saber a través de

qué pasos se ha llegado a esa instrucción. Finalmente, en la pestaña **Observar** podemos introducir cualquier expresión en BASIC, incluyendo operaciones con las variables que el programa tenga declaradas para ver cuál es el resultado o el valor que tienen en el momento de la pausa.

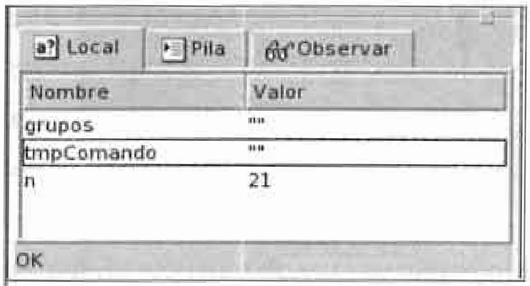


Figura 3. Pestañas **Local**, **Pila** y **Observar** de la ventana **Proyecto**.

2. 6 Entrada y salida: ficheros

En este apartado veremos la forma más común de trabajar con ficheros en Gambas. Gambas trata los ficheros como un flujo de datos (la palabra exacta para esto es *Stream*), lo que tiene una implicación muy cómoda: todos los flujos de datos se tratan de igual manera, con lo que el código para manejar un fichero es igual al código para manejar una conexión de red o un puerto de comunicaciones, puesto que todos son objetos de tipo *Stream*. Las operaciones tradicionales con un fichero son abrirlo, crearlo, escribir y leer datos. Veamos cómo se usan:

```
Archivo = OPEN Nombre_de Archivo FOR [ READ | INPUT ]
[ WRITE | OUTPUT ] [ CREATE | APPEND ] [ WATCH ]
```

Abrimos un archivo con distintas finalidades:

- READ o INPUT: para leer datos, en el primer caso no se usa buffer de datos, con INPUT sí que hay un buffer intermedio.

- WRITE o OUTPUT: para escribir datos, con WRITE no hay buffer de datos, con OUTPUT sí se usa.

- CREATE: si el fichero no existe se crea. Si no se usa esta palabra, el fichero debe existir antes de abrirlo o dará un error.

- APPEND: los datos se añaden al final del fichero.

- WATCH: si se especifica esta palabra, Gambas lanzará los eventos (que veremos más adelante) *File_Read* y *File_Write* en caso de que se pueda leer y escribir en el archivo.

Ahora, cerremos un **Archivo** que ha sido abierto con la sentencia **OPEN**.

```
CLOSE [ # ] Archivo
```

Escribimos, convirtiendo en cadena de texto, la **Expresión** en el **Archivo** abierto anteriormente.

```
PRINT [ #Archivo , ] Expresión ]
```

Si no se especifica ningún archivo, tecleamos la expresión en la consola, como se ha visto en distintos ejemplos a lo largo de todo este capítulo. La instrucción PRINT admite un cierto control de cómo se colocan las expresiones, dependiendo de algunos signos de puntuación que se pueden colocar al final de la sentencia:

- Si no hay nada después de la *Expresión*, se añade una nueva línea al final. Por tanto, la salida de la siguiente instrucción *PRINT* será en una línea nueva.

- Si hay un punto y coma detrás de la *Expresión*, la siguiente instrucción *PRINT* se escribirá justo detrás de la salida anterior, sin espacios, líneas o tabulaciones intermedias.

- Si hay un punto y coma doble, se añade un espacio entre las expresiones, lo que permite concatenar expresiones en una misma línea usando distintas sentencias *PRINT*.

- Si se utiliza una coma en lugar de un punto y coma, se añade una tabulación, con lo que también se pueden concatenar expresiones en una misma línea.

Seguidamente, escribimos, sin convertir en cadena de texto, la **Expresión** en el **Archivo** abierto anteriormente. Es una instrucción que suele usarse en lugar de *PRINT* cuando los datos a escribir no son cadenas de texto, como en el caso de archivos binarios.

```
WRITE [ #Archivo , ] Expresión [ , Longitud ]
```

En cualquier caso, también puede usarse con cadenas de texto y permite indicar, con el parámetro **Longitud**, el número de caracteres que se desean sacar en cada operación de escritura. Al contrario que con *PRINT*, no se pueden usar signos de puntuación para controlar la posición de la escritura.

```
INPUT [ #Archivo , ] Variable1 [ , Variable2 ... ]
```

Leemos, desde un **Archivo**, un dato y asignamos su valor a **Variable1**, **Variable2**, etc. Los datos deben estar separados en el archivo por comas o en distintas líneas. Si no se especifica el **Archivo**, leemos los datos desde la consola, esperando que el usuario de la aplicación los introduzca.


```
READ [ #Archivo , ] Variable [ , Longitud]
```

Se puede decir que es la instrucción opuesta a *WRITE*. Leemos del **Archivo** datos binarios y les asignamos su valor a la **Variable**. Si ésta es una cadena de texto, se puede fijar la **Longitud** de la cadena a leer.

```
LINE INPUT [ #Archivo , ] Variable
```

En una línea de texto entera del **Archivo**, le asignamos a la **Variable**. No se debe usar para leer de flujos binarios.

```
Eof ( Archivo )
```

Devuelve *True* (verdadero) cuando se llega al final del **Archivo** y *False* (falso) en caso contrario.

```
Lof ( Flujo )
```

Si el **Flujo** de datos es un archivo, devuelve su tamaño. Si en lugar de un archivo es un *Socket* de una conexión de red o un objeto *Process*, devuelve el número de bytes que pueden leerse de una sola vez. Una vez vistas las sentencias más comunes para el manejo de flujos de datos, veamos algunos ejemplos de uso:

```
' Leer datos de un puerto serie:
' (Requiere seleccionar el componente gb.net en el
proyecto)
DIM Archivo AS File
Archivo = OPEN "/dev/ttyS0" FOR READ WRITE WATCH
...
' El evento File_Read se produce cuando haya datos
que leer:
PUBLIC SUB File_Read()
  DIM iByte AS Byte
  READ #Archivo, iByte
  PRINT "Tengo un byte: "; iByte
END
```

```
'Lee el contenido del fichero /etc/passwd y lo
muestra en la consola:
DIM Archivo AS File
DIM Linea AS String
Archivo = OPEN "/etc/passwd" FOR INPUT

WHILE NOT Eof(Archivo)
  LINE INPUT #Archivo,  Linea
  PRINT  Linea
WEND
CLOSE Archivo
```

2. 7 Control de errores

Es seguro que, en algún momento, en la mayor parte de los programas, bien por acciones del usuario del programa, bien por el propio flujo de la ejecución, se producen errores, como intentar borrar un fichero que no existe, hacer una división por cero, conectar a un servidor web que no responde, etc. En todos estos casos, Gambas muestra un mensaje en pantalla y, a continuación, el programa se rompe y deja de funcionar. Es evidente que ése es un comportamiento que el desarrollador no desea y debe poner las medidas oportunas para evitarlo. La forma de hacerlo es implementando un control de errores para que la aplicación sepa lo que debe hacer en esos casos. Gambas implementa las instrucciones necesarias para capturar los errores y procesarlos según los deseos del programador. Las instrucciones para ello son:

- *TRY Sentencia*: ejecuta la sentencia sin lanzar el error aunque se produzca, el programa continúa por la instrucción que haya después del *TRY*, tanto si hay error como si no. Se puede saber si existe error consultando la sentencia *ERROR* que valdrá verdadero o falso. Por ejemplo:

```
  ' Borrar el archivo aunque no exista
TRY KILL "/tmp/prueba/"
' Comprobar si ha tenido éxito
IF ERROR THEN PRINT "No fue posible eliminar el archivo"
```

- *FINALLY*: se coloca al final de un procedimiento. Las instrucciones que siguen a continuación se ejecutan siempre, tanto si ha habido un error en el procedimiento como si no lo ha habido.

- *CATCH*: se coloca al final de un procedimiento. Las instrucciones que siguen a continuación se ejecutan sólo si se ha producido un error en la ejecución del procedimiento (incluyendo si el error se ha producido en subrutinas o funciones llamadas desde el mismo procedimiento). Si existe una instrucción *FINALLY*, ha de colocarse delante de *CATCH*.

Veamos un ejemplo de FINALLY y CATCH:

```
' Mostrar un archivo en la consola
SUB PrintFile(Nombre_Archivo AS String)
DIM Archivo AS File
DIM Linea AS String

OPEN Nombre_Archivo FOR READ AS #Archivo

WHILE NOT EOF(Archivo)
  LINE INPUT #Archivo,  Linea
  PRINT  Linea
WEND

FINALLY ' Siempre se ejecuta, incluso si hay error
  CLOSE # Archivo

CATCH ' Se ejecuta sólo si hay error
  PRINT "Imposible mostrar el archivo ";  Nombre_Archivo
END
```


2. 8 Programación orientada a objetos con Gambas

BASIC es un lenguaje de programación estructurada. Esto significa que los programas tienen el flujo que hemos visto hasta ahora: empiezan en un punto de una subrutina y van ejecutando las instrucciones de arriba a abajo, con los saltos correspondientes a las llamadas a procedimientos y distintas funciones de control del flujo como bucles, condiciones, etc.

Este tipo de programación permite resolver la mayor parte de los problemas y, de hecho, se ha estado usando durante muchos años. A día de hoy, se siguen desarrollando

aplicaciones con lenguajes, como BASIC o C, de programación estructurada. Sin embargo, desde finales de los años 70 se ha venido trabajando también en otro paradigma de programación: la programación orientada a objetos. Los lenguajes que adoptan este paradigma, como Smalltalk, Java o C++, intentan modelar la realidad. La ejecución de estos programas se basa en la interacción de los distintos objetos que definen el problema, tal y como ocurre en la vida normal, en la que los objetos, personas y animales nos movemos, enviamos mensajes unos a otros y ejecutamos acciones. La programación orientada a objetos se usa cada día con más frecuencia porque permite una mejor división de las tareas que un programa debe hacer, facilita la depuración y la colaboración entre distintos programadores en los proyectos que son de gran tamaño y, en muchos aspectos, tiene un potencial mucho mayor que la programación estructurada.

Por todas estas razones, es común que se añadan características de programación orientada a objetos a lenguajes que, como BASIC, en un principio no la tenían. Gambas permite hacer programas estructurados a la vieja usanza, escribiendo el código en *Módulos*, como hemos visto. Y también posibilita la programación orientada a objetos mediante el uso de *Clases*. Es más, para la programación gráfica y el uso de la mayor parte de los componentes que se añaden al lenguaje, no hay más alternativa que el uso de objetos.

Podemos ver cómo funciona todo esto en Gambas pensando en el caso de que tuviéramos que escribir un programa que simulara el comportamiento de un coche. Usando la programación orientada a objetos, definiríamos cada una de las partes del coche mediante un archivo de clase en el que escribiríamos el código BASIC necesario para definir las características de esa parte y la interfaz con la que se comunica con el mundo real. Por ejemplo, definiríamos cómo es un volante, cómo es una rueda, cómo es un asiento, un acelerador, un motor, etc. Después, y basándonos en esa definición, crearíamos cada uno de los objetos para crear el coche: un volante, cuatro ruedas, varios asientos, un motor, un acelerador, etc. Cada uno de esos objetos respondería a ciertos mensajes. Por ejemplo, el volante respondería a un giro actuando sobre el eje de las ruedas, el motor respondería incrementando sus revoluciones si recibe una presión del acelerador, etc.

Cada vez que creamos un objeto basándonos en el archivo de clase que lo ha definido, decimos que el objeto ha sido *instanciado*. Se pueden instanciar tantos objetos como se desee a partir de una clase y una vez creados tienen vida propia, son independientes unos de otros con sus propias variables internas y respondiendo a las distintas acciones según hayan sido definidos todos en la clase.

Otra de las características de la programación orientada a objetos es la *Herencia*. Cuando un objeto hereda de otro objeto significa que es del mismo tipo, pero que puede tener características añadidas. Por ejemplo, supongamos que definimos la clase *cuadro_de_texto* con ciertas características como tamaño de texto, longitud, etc. A continuación, podemos crear objetos de esa clase y son *cuadros_de_texto*. Con Gambas podemos, además, crear una nueva clase, por ejemplo: *cuadro_de_texto_multilínea* que herede de *cuadro_de_texto*. Eso significaría que un *cuadro_de_texto_multilínea* es un *cuadro_de_texto* al que se le añaden más cosas. Todo el comportamiento y propiedades del *cuadro_de_texto* ya están codificados y no hay que volver a hacerlo.

Veamos un ejemplo sencillo que aclare algunos conceptos, para lo que vamos a crear en el entorno de desarrollo un nuevo proyecto de programa de texto. A continuación, le añadimos un *Módulo* con un nombre cualquiera y dos archivos de clase, a uno lo llamamos *SerVivo* y a otro *Hombre*.

Éste es el código que debemos escribir en el archivo de clase *SerVivo.cls*:

```
' Gambas class file
PRIVATE patas AS Integer
PRIVATE nacimiento AS Integer
PUBLIC SUB nacido(fecha AS Date)
  nacimiento = Year(fecha)
END

PUBLIC SUB PonePatas(numero AS Integer)
  Patas = numero
END
```

```
PUBLIC FUNCTION edad() AS Integer
  RETURN Year(Now) - nacimiento
END

PUBLIC FUNCTION dicePatas() AS Integer
  RETURN patas
END
```

Éste es el código a escribir en el archivo de clase *Hombre.cls*:

```
' Gambas class file
INHERITS SerVivo
PRIVATE Nombre AS String
PRIVATE Apellido AS String

PUBLIC SUB PoneNombre(cadena AS String)
  Nombre = cadena
END

PUBLIC SUB PoneApellido(cadena AS String)
  Apellido = cadena
END

PUBLIC FUNCTION NombreCompleto() AS String
  RETURN Nombre & " " & Apellido
END
```

Y éste el código a escribir en el Módulo que hayamos creado:

```
' Gambas module file
PUBLIC SUB Main()
DIM mono AS SerVivo
DIM tipejo AS Hombre
```

```
mono = NEW SerVivo
mono.nacido(CDate("2/2/1992"))
mono.PonePatas(3)
PRINT mono.edad()
PRINT mono.dicePatas()

tipejo = NEW hombre
tipejo.nacido(CDate("2/18/1969"))
tipejo.PoneNombre("Vicente")
tipejo.PoneApellido("Pérez")
PRINT tipejo.edad()
PRINT tipejo.NombreCompleto()
END
```

Veamos los tres archivos, empezando con el de clase *SerVivo.cls*. Tiene un código muy simple: declara un par de variables, un par de subrutinas con las que se puede asignar el valor a esas variables y dos funciones con las que devuelve valores de algunos cálculos realizados sobre las variables y la fecha actual.

El archivo *Hombre.cls* es muy similar, pero con una sentencia nueva: al inicio del fichero se ha colocado la instrucción **INHERITS SerVivo**. Al haber hecho eso estamos diciendo que todos los objetos que se instancien de la clase *Hombre* serán objetos *SerVivo* y, por tanto, tendrán también las mismas funciones que un *SerVivo* y podrán realizar las mismas operaciones.

Si hubieran sido necesarias algunas labores de inicialización de los objetos al ser creados, se harían añadiendo al archivo de clase una subrutina con la sintaxis: *PUBLIC SUB _New()*. Todo el código que se escriba ahí se ejecutará cada vez que se cree un objeto.

Y, finalmente, el módulo y su procedimiento **Main**. Ahí se declaran dos variables: **mono** y **tipejo**, pero en lugar de declararlos como uno de los tipos de datos que vimos en los primeros apartados de este capítulo, se declaran como objetos de la

clase *SerVivo* y *Hombre*. Por tanto, hablando con propiedad, **mono** y **tipejo** no son variables, sino objetos. Las clases corresponden a los nombres de los dos archivos de clase que hemos definido. Además, vemos que para crear un objeto de una clase es necesario usar siempre la palabra **NEW**, lo que provocará además que se ejecute el código de la subrutina *_New* del archivo de clase, si esta subrutina existiera. Una vez que el objeto ha sido creado, podemos acceder a sus subrutinas y funciones escribiendo el nombre de un objeto y el procedimiento separados por un punto. Sólo podemos acceder a los procedimientos que hayan sido declarados en el archivo de clase como **PUBLIC**.

Aparte del ahorro de código obvio que supone para la clase *Hombre* no tener que escribir las funciones que hace un *SerVivo*, hay algunas consecuencias más importantes que se deducen de este ejemplo. Tal y como está escrito, se podría alterar el código de la clase *SerVivo*, modificando y cambiando variables, y el programa seguiría funcionando igual, sin tener que tocar en absoluto la clase *Hombre*.

Es decir, se puede, por ejemplo, mejorar el método para calcular la edad, teniendo en cuenta el día de nacimiento dentro del año. También se puede cambiar el nombre de las variables y no afecta a la clase *Hombre* y al resto del programa. Otra de las posibilidades que existe es el uso de los archivos de clase tal y como están en otro proyecto donde, del mismo modo, se necesiten seres vivos, reutilizando de forma sencilla el código ya escrito.

Se han elaborado multitud de libros y artículos sobre la programación orientada a objetos. La pretensión de este apartado es únicamente servir de mera introducción general y explicar la sintaxis necesaria para su uso con Gambas. La mejor forma de aprender, disfrutar de ella y obtener todo su potencial es probando y viendo ejemplos que la usen extensivamente.

El código fuente del entorno de desarrollo de Gambas hace un uso muy amplio de clases y objetos, con lo que es una fuente completa de ejemplos. Está disponible dentro del directorio *apps/src/gambas2* del fichero comprimido que contiene los fuentes de Gambas.

2. 9 Propiedades, Métodos y Eventos

Los componentes que extienden las posibilidades de Gambas son de distintos tipos, pero todos ellos contienen clases que definen distintos objetos. En el caso de los componentes que sirven para crear interfaces gráficas, estos objetos incorporan un icono a la **Caja de Herramientas** del entorno de desarrollo. No hay que crear este tipo de objetos escribiendo el código correspondiente y la palabra *NEW*, sino sólo dibujándolos sobre un formulario después de haber pulsado el botón con el icono respectivo. Es el entorno de desarrollo el que se encarga de escribir el código necesario para crearlos.

Cualquiera de los objetos que se crean a partir de los componentes tienen ya un comportamiento definido en su clase. Este comportamiento incluye la interfaz con la que los objetos se van a comunicar unos con otros en el programa, y lo necesario para que el programador pueda definirlos y hacerlos actuar a su conveniencia. Para facilitar esas tareas, estos objetos disponen de *Propiedades*, *Métodos* y *Eventos*. Las *propiedades* permiten cambiar parámetros del objeto. En realidad, al darle un valor a una propiedad no estamos sino asignando valores a algunas variables del objeto que éste interpreta internamente para producir un efecto. Por ejemplo, podemos asignarle a un formulario un valor numérico a la propiedad *Background*, y con ello le cambiaremos el color del fondo. Para modificar las propiedades disponemos de la ventana de **Propiedades** en el IDE (visible al pulsar **F4** o desde el menú de la ventana de **Proyecto: Vista | Propiedades**), que nos permite hacerlo con una interfaz gráfica sencilla. También podemos cambiar una propiedad mediante código haciendo referencia al nombre del objeto y a la propiedad, separados por un punto. Por ejemplo: *Form1.Background=0*, dejaría el fondo del formulario *Form1* en color negro.

Los *métodos* son las funciones que el objeto puede realizar. Sólo pueden asignarse mediante código y cada objeto tiene su propia colección de métodos. Por ejemplo: *btnSalir.Hide* haría que el botón *btnSalir* se ocultara. Finalmente, los *eventos* son subrutinas que se ejecutan para indicar algo que le ha ocurrido al objeto. Un ejemplo que ya vimos en el capítulo anterior es el evento *Click* del botón que se ejecuta cuando hacemos clic con el ratón. Los eventos son, en las aplicaciones gráficas, los que marcan el flujo del programa. De hecho, con frecuencia se dice que Gambas es

un lenguaje de programación orientado a eventos. Es el usuario de la aplicación al interaccionar, el que obliga al programa a ejecutar código respondiendo a sus acciones. Los objetos en Gambas avisan con los eventos cada vez que se produce una acción sobre ellos. Es el programador el encargado de escribir en las subrutinas que tratan los eventos el código necesario para responder a ellos.

El entorno de desarrollo proporciona varias formas de conocer los métodos, eventos y propiedades que un objeto puede entender. La primera y más obvia es usando la ayuda, donde se puede encontrar el listado de todas las clases de objetos disponibles, con todas las explicaciones y el listado completo de las propiedades. El editor de código también suministra información, puesto que al escribir el nombre de un objeto y pulsar un punto, aparece un menú contextual con el listado de las propiedades (en color violeta) y métodos (en color verde) que el objeto tiene disponibles.

Para los objetos gráficos hay algunas ayudas más. El listado de propiedades disponible se puede ver en la ventana de **Propiedades.** Para los eventos podemos hacer clic con el botón derecho sobre un objeto gráfico en su formulario. Aparecerá un menú donde, entre otras, vemos la opción **Evento.** Podemos elegir el evento que queramos y, automáticamente, el entorno ya escribirá el código correspondiente al principio y al final del procedimiento.



Figura 4. Evento Click.

NOTAS

[1] Los ordenadores sólo entienden de números, no de letras. Para poder usar caracteres se aplican tablas de conversión que asignan un número o código a cada carácter. La tabla de conversión usada normalmente se llama ASCII.
En **http://es.wikipedia.org/wiki/Ascii** se puede ver la tabla completa.

3. La interfaz gráfica

3. La interfaz gráfica

3. La interfaz gráfica

3. La interfaz gráfica

3.1 Concepto

Si bien Gambas, como cualquier otro lenguaje de programación bien diseñado, puede trabajar perfectamente desligado de toda librería gráfica, creando programas de consola, uno de sus puntos fuertes es la sencillez para crear interfaces gráficas de usuario.

En el mundo GNU/Linux ha existido, a lo largo de la historia, diversas librerías que facilitan la creación de dichas interfaces. Al nivel más bajo, el tradicional sistema X-Window tan sólo proporciona una API para mostrar ventanas, dibujar líneas, copiar mapas de bits y poco más. Sobre dicho sistema, una de las primeras librerías de apoyo que proporcionaba controles completos, tales como botones o etiquetas, fue la Motif™, tradicional de sistemas UNIX™. Su clon libre, Lesstif, probablemente llegó demasiado tarde al escenario para tener un papel relevante.

Las interfaces creadas con estas dos librerías han tenido siempre fama, además, de ser bastante feas o incómodas, especialmente a ojos de los usuarios de Windows™, y no hay que olvidar que de este sistema propietario proviene buena parte de los actuales usuarios de escritorio GNU/Linux.

La compañía TrollTech™, por su parte, creo las librerías QT, para el desarrollo de aplicaciones gráficas con C++. Estas librerías, en principio, se distribuían bajo una licencia, la QPL, no totalmente compatible con el proyecto de la Free Software Foundation. Su inclusión como base del proyecto de escritorio KDE generó un gran revuelo, y el rechazo de un sector de la comunidad hacia ambos proyectos (QT y KDE). En la actualidad, las librerías QT en su edición no comercial, se distribuyen bajo licencia GPL, lo cual implica que los programas desarrollados y compilados con QT como base, han de ser también Software Libre compatible con la GPL. Un programa que no cumpla esta norma, habría de ser compilado sobre la versión comercial de QT, que la compañía antes citada vende y soporta.

En parte como rechazo al tándem QT/KDE, y en parte para crear una alternativa al popular escritorio KDE, surgió el proyecto GNOME, que está basado en las librerías GTK+.

GTK+ se desarrolló al principio como una librería gráfica escrita únicamente para el popular programa de dibujo The Gimp (de hecho, GTK significa *Gimp Tool Kit*). Pero más tarde se escindió de este proyecto para convertirse en una librería de propósito general, especialmente diseñada para desarrollos en lenguaje C. Hoy en día, todo el proyecto GTK+ está dividido en varios bloques y niveles: Glib, utilidades de carácter general sin relación con la interfaz gráfica: Gobject para dotar, de cierta orientación a objetos al lenguaje C; Atk, que es un kit de accesibilidad; Pango, para la gestión de fuentes; Gdk, para el dibujo de bajo nivel; y, finalmente, GTK, que proporciona los elementos de la interfaz gráfica habitual. La licencia de GTK+ es LGPL, por lo que ha sido utilizada, además de en muchos proyectos de Software Libre, en programas gráficos privativos que no desean contar con el soporte de la versión comercial de QT porque la han visto como una alternativa más cómoda en sus desarrollos, o han decidido reducir los costes al no tener que pagar por la librería gráfica.

Al margen de estos pesos pesados, hay nombres como FOX, FLTK o WxWidgets, que también resuenan como alternativas para el desarrollo de programas gráficos.

Por tanto, hay muchas alternativas (toolkits) para desarrollar interfaces, y al menos dos de ellas (QT y GTK+) sirven como base para los dos escritorios más comunes (KDE y GNOME), que a su vez aportan un aspecto y funcionalidades diferentes. No obstante, una aplicación KDE puede funcionar en un entorno GNOME y viceversa, a costa, quizá, de perder homogeneidad en el entorno.

Gambas ha decidido ser neutral al respecto. Aporta una interfaz de alto nivel, sencilla para el diseño y programación habituales, que no está ligada a los conceptos de QT, GTK+, ni a ninguna otra librería gráfica subyacente.

No obstante, a la hora de implementar dicha interfaz sí que es necesario emplear alguna librería de sustento por debajo, escrita en C o C++. Por tanto, existen dos componentes gráficos que proporcionan al programador libertad de elección: *gb.qt* y *gb.gtk*. Como se puede suponer por sus nombres, el primero utiliza código compilado con QT y el segundo código compilado con GTK+.

La particularidad de Gambas es que el código escrito para *gb.qt* funcionará exactamente igual si reemplazamos este componente por *gb.gtk* y viceversa. Por lo tanto, el programador en cada momento puede elegir el que más se adapte a sus necesidades por diversos motivos, por ejemplo:

- Integración con KDE, GNOME o XFCE (este último, un escritorio ligero basado en GTK+).

- Aspecto final. Algunos programadores y usuarios se sienten más a gusto con una aplicación QT, otros con GTK+.

- Cuestiones de rendimiento, uso de recursos.

- Licencia, costes en software comercial.

- Necesidades especiales. GTK+ se puede compilar sobre DirectFB, un sistema gráfico alternativo a X-Window, ligero y apto para sistemas empotrados. Tal vez *gb.qt* se pueda compilar en el futuro sobre Qtopia, la librería de TrollTech™ generalmente empleada en PDAs.

□□□□□ Partiendo de la consola

Vamos a empezar por el camino difícil para comprender cómo la interfaz gráfica no es más que otro accesorio de Gambas, que al igual que el resto de componentes, aporta clases a partir de las cuales crearemos objetos, en este caso *Controles* o *Widgets*.

Crearemos un proyecto de consola, (sí, de consola), llamado **Ventana**. Añadiremos un módulo **modMain** y una referencia al componente **gb.qt**.

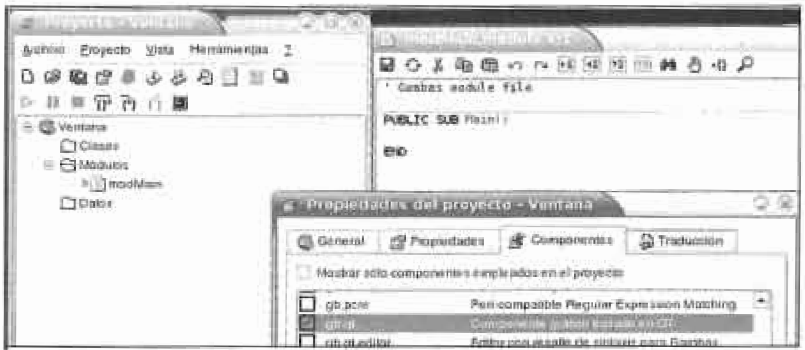


Figura 1. Proyecto **Ventana**.

Escribiremos ahora el siguiente código:

```
PUBLIC SUB Main()

  DIM hWin AS Window

  hWin = NEW Window
  hWin.Show()

END
```

Al ejecutarlo, aparece una ventana solitaria en la pantalla, que desaparecerá cuando pulsemos el botón **Cerrar** del gestor de ventanas.

Las ventanas son contenedores de primer nivel. Un *contenedor* es un control que permite situar otros en su interior (botones, cuadros de texto, etc.). Y el adjetivo de *primer nivel* se refiere a que no tiene un *padre*, es decir, que no cuelga de otro control de nuestra aplicación, sino directamente del escritorio.

En este programa podemos ver varios efectos, a primera vista, extraños. El primero es que hemos declarado **hWin** como una variable local, así pues, parece que al finalizar la función **Main** el objeto debería destruirse. Esto no así ya que la ventana, al ser un control, mantiene una referencia interna (los objetos Gambas sólo se destruyen si no existe una referencia en todo el programa). Dicha referencia podemos decir que se corresponde con la ventana que está dibujada en el servidor gráfico, con la intermediación de la librería gráfica (en este caso QT). Por otra parte, el programa debería haber finalizado al terminar la función **Main()** y no ha sido así.

El segundo efecto se debe a que tanto el componente *gb.qt* como el componente *gb.gtk* son llamados automáticamente tras el método **Main()** del programa, y queda en el lazo principal de la librería gráfica, esperando a que se produzcan eventos gráficos (por ejemplo, una pulsación del ratón sobre un control).

En la práctica, para crear un programa gráfico indicaremos directamente en el asistente que deseamos crear un proyecto gráfico, de modo que el entorno de desarrollo incluye ya, por defecto, el componente *qb.qt*, y nos permite crear un *formulario de inicio*. Un formulario de inicio es una clase de inicio que hereda las características de la clase *Form*, que no necesita de un método *Main* en el programa porque el intérprete llama directamente al formulario para mostrarlo, y entra en el lazo de gestión de eventos.

□□□□□ El entorno de desarrollo

Para crear un nuevo formulario nos situaremos en la ventana principal del IDE y, tras pulsar el botón derecho, elegiremos **Nuevo | Formulario**.

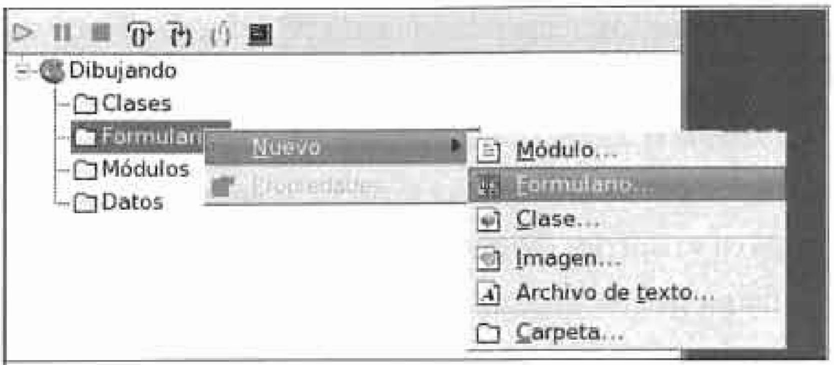


Figura 2. Pasos para crear un nuevo formulario.

Ahora pasaremos a situar los diferentes controles, para lo cual los seleccionaremos de la Caja de herramientas y los posicionaremos sobre los formularios manteniendo el botón izquierdo pulsado, mientras le damos el tamaño deseado. El código se irá escribiendo en función de los eventos que generen los controles, por ejemplo, el evento *Click* de los botones o los eventos de ratón y teclado de cada control.



Figura 3. Controles disponibles en la **Caja de herramientas**.

Al pulsar con doble clic sobre un control, se genera, automáticamente, el encabezado del código correspondiente al evento por defecto (el más característico) del control seleccionado.

Además, pulsando con el botón derecho se puede seleccionar la lista de eventos disponibles para un control.

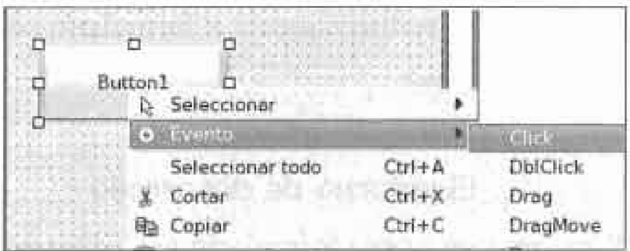


Figura 4. Eventos del control **Button1**.

3. 2 Manejo básico de los controles

A pesar de que cada control ha sido diseñado para cumplir una función específica, comparten buena parte de su interfaz de programación, de modo que aprender a manejar un nuevo control sea tarea sencilla para un programador que ya ha trabajado con otros controles.

Los controles heredan métodos, propiedades y eventos de la clase *Control*, por lo que todas las características heredadas son aplicables a todos ellos.

En cuanto a los contenedores, proceden de la clase *Container* (que a su vez procede de *Control*) e igualmente tienen muchas características comunes.

Posición y tamaño

Todos los controles disponen de una serie de propiedades que permiten conocer y modificar su posición y tamaño dentro de su contenedor, o del escritorio en el caso de las ventanas:

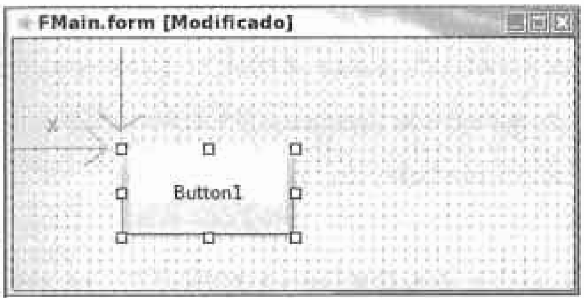


Figura 5. Propiedades *X* e *Y* del control.

- **Propiedades X e Y**: son de lectura y escritura, y determinan la posición del control, es decir, su punto superior izquierdo. En los controles comunes, la posición indicada es relativa a su contenedor, y en el caso de las ventanas es relativa a la esquina superior izquierda del escritorio. Los controles disponen de otras dos propiedades, *Left* y *Top*, que son sinónimas de *X* e *Y*, respectivamente. Usar una u otra se deja a la elección del programador.

- **Propiedades W y H**: son de lectura y escritura, y determinan el ancho y alto del control, respectivamente. Dispone de dos propiedades sinónimas. *Width* y *Height* con el mismo significado.

- **Propiedades ScreenX y ScreenY**: son de sólo lectura, y permiten conocer la posición de cualquier control relativa al escritorio, en lugar de a su contenedor *padre*.

Los contenedores disponen, además, de las propiedades *ClientX*, *ClientY*, *ClientWidth* y *ClientHeight*, que determinan, respectivamente, el inicio y la dimensión del área útil para contener controles *hijo*. Por ejemplo, el control *TabStrip*, que dispone de unas pestañas en la parte superior, sitúa a sus *hijos* por debajo de ellas; y *ScrollView*, que puede mostrar barras de Scroll, ve su área útil reducida por dichas barras.

Existen también una serie de métodos para modificar los controles:

- **Método Resize(W,H)**: con él podemos cambiar el tamaño de un control modificando su alto y ancho de una sola vez, en lugar de hacerlo en dos pasos modificando las propiedades *W* y *H*, lo que mejora el efecto gráfico de la redimensión ante el usuario.

- **Método Move(X,Y)**: mueve de una sola vez el control a la posición indicada, en lugar de hacerlo en dos pasos. También dispone de dos parámetros adicionales, *Move(X,Y,W,H)*, con los cuales además de mover el control se puede redimensionar, todo ello en un solo paso, generando una transición más suave ante el usuario, que si modificáramos las propiedades una por una.

- **Métodos MoveRelative y ResizeRelative**: son similares a *Move* y *Resize*, respectivamente, pero en este caso las unidades no son píxeles, sino unidades relativas al tamaño de la fuente por defecto del escritorio. Con esta capacidad, el aspecto del formulario será similar para usuarios que utilicen distintas configuraciones de fuentes (por ejemplo, grandes en un escritorio a 1024x768 o más pequeñas en un escritorio a 800x600).

Las ventanas (controles *Window* y *Form*) disponen, por su parte, de varios eventos relacionados con la posición y tamaño. El evento Resize se genera cada vez que el usuario redimensiona una ventana, y *Move* cuando se mueve.

Visibilidad

Todo control puede encontrarse en uno de estos dos estados en un momento dado: *Visible*, que significa que aparece representado ante el usuario; e *Invisible*, que mantiene todas sus propiedades, pero no aparece de cara al usuario.

La propiedad *Visible* sirve para conocer el estado de visibilidad, en todo momento.

Por ejemplo, los métodos *Show()* y *Hide()* muestran y ocultan, respectivamente, el control.

Con lo que respecta a las ventanas, disponen de dos eventos, *Show* y *Hide*, que se disparan cuando se muestran o dejan de estar visibles.

Además, la apertura por primera vez de un formulario o ventana genera el evento *Open*.

Crear efectos de *blinking*, controles tales como etiquetas que aparecen y desaparecen para llamar la atención del usuario, puede ser útil en alguna ocasión especial, si se reclama atención por algún motivo crítico. Pero, en general, abusar de esta técnica molesta a los usuarios y se considera poco apropiada en una interfaz correctamente diseñada.

Textos relacionados

Todos los controles pueden tener diversas cadenas de texto, que muestran de una u otra forma.

Para todos ellos, la propiedad *ToolTip* se encarga de mantener un texto que aparecerá como una pequeña ventana flotante, cuando el usuario sitúe el ratón por encima del área ocupada con el control.

De esta forma, se pueden dar pequeñas informaciones de guía para que el usuario conozca la función de dicho control dentro del programa.

Muchos controles disponen de un texto que muestran en su espacio principal, como es el caso de las etiquetas, las cajas de texto o los botones. Para todos estos controles, la propiedad *Text* es la que determina la cadena a mostrar.

Puesto que Gambas trabaja con codificación UTF-8 para la interfaz gráfica, puede que sea necesario utilizar la función *Conv$* para convertir desde otras codificaciones, a la hora de representar texto procedente, por ejemplo, de una base de datos o de un proceso en ejecución en segundo plano.

Estos controles, por comodidad para programadores de otros entornos, tienen un sinónimo para esta propiedad, denominado *Caption*, salvo en el caso de las ventanas cuyo sinónimo es *Title*.

Colores

Para cada control se definen dos colores: *ForeGround*, que es el color de primer plano en el que normalmente se mostrará el texto del control o parte de sus líneas y dibujos; y *Background*, que es el color de fondo.

Por herencia de otros lenguajes de programación, disponen de dos sinónimos: *BackColor* y *ForeColor*.

Los colores en Gambas son valores numéricos enteros, desde el 0 al hexadecimal FFFFFF, de forma que cada componente de color viene determinado por sus componentes de rojo, verde y azul.

La intensidad de cada uno de estos colores básicos varía entre el 0 (mínimo) y el 255 (máximo).

La gestión del color se realiza en Gambas a través de la clase *Color*, que dispone de varios métodos estáticos, así como de una serie de constantes.

Las constantes de la clase *Color* determinan una serie de colores básicos, en su codificación numérica:

COLOR	TONALIDAD NORMAL	TONALIDAD OSCURA
Negro	Color.Black	—
Blanco	Color.White	—
Azul	Color.Blue	Color.DarkBlue
Cian	Color.Cyan	Color.DarkCyan
Magenta	color.Magenta	Color.DarkMagenta
Naranja	color.Orange	—
Rosa	color.Pink	—
Rojo	color.Red	Color.DarkRed
Violeta	color.Violet	—
Amarillo	color.Yellow	Color.DarkYellow
Gris	Color.Gray	—
Verde	Color.Green	Color.DarkGreen
Gris Claro	Color.LightGray	—

El método RGB recibe tres parámetros, las componentes de rojo, verde y azul, en este orden, con valores entre 0 y 255 para cada uno, y devuelve un número representando al color.

El método HSV recibe tres parámetros, las componentes de tonalidad (0-360), saturación (0-255) y brillo (0-255), en este orden, y lo devuelve traducido a un color con su codificación numérica habitual.

Existen también unas propiedades que determinan los colores del sistema.

Por ejemplo, y dependiendo del tema que haya seleccionado el usuario, los cuadros de texto pueden aparecer de color blanco con letra negra, o los formularios de color gris.

Estas propiedades permiten conocer los colores actuales para los elementos de la interfaz gráfica:

Color de fondo general	Color.Background
Color de fondo de los botones	Color.ButtonBackground
Color de primer plano de los botones	Color.ButtonForeground
Color de primer plano general	Color.Foreground
Color de fondo de un elemento seleccionado	Color.SelectedBackground
Color de primer plano de un elemento seleccionado	Color.SelectedForeground
Color del texto en cajas de texto	Color.TextBackground
Color de fondo en cajas de texto	Color.TextForeground

Es poco recomendable cambiar los colores de la interfaz por capricho o cuestiones de estética particulares del programador. Cada usuario elige el tema que más se adapta a sus gustos o necesidades visuales, y puede sentirse incómodo si le forzamos a usar otros colores. Por otra parte, algunos temas pueden interferir sus colores con los que hemos definido en nuestra aplicación, resultando la combinación difícil de ver, molesta o desagradable. Sólo se cambiarán colores de los controles, cuando sea expresamente necesario por algún motivo de diseño (por ejemplo, resaltar de forma clara un texto en una etiqueta).

□□□□□ Ratón

El ratón es la interfaz por excelencia de cualquier escritorio actual. Hemos de distinguir aquí dos apartados.

En primer lugar se encuentra la representación de éste en el escritorio, el puntero, que habitualmente aparece como una flecha, blanca o negra. Al respecto, cada control dispone de una propiedad *Mouse*, la cual puede tomar como valores las constantes de la clase *Mouse* para cambiar su aspecto.

Se consideran constantes para que el ratón adopte diversos aspectos, como puede ser un reloj (espera), un cursor de texto, flechas en diversas orientaciones, etc.

Las constantes de la clase *Mouse* son las mismas para *gb.qt y gb.gtk*, no obstante, sus valores numéricos son distintos. Por tanto, un código bien escrito y escalable, no debe usar valores numéricos para indicar un tipo de puntero, sino las constantes de esta clase.

Cada control dispone, además, de una propiedad *Cursor*, que acepta una imagen y permite dibujar un puntero totalmente personalizado (a partir de un archivo *.png* o *.xpm*, por ejemplo) para cada control. Dependiendo del servidor gráfico utilizado en el sistema, es posible que el cursor pueda tener también varios colores, no sólo en blanco y negro como los tradicionales.

La clase estática *Application*, aporta una propiedad *Busy*, que es un número entero. Si su valor es mayor que cero, todos los controles y ventanas de la aplicación mostrarán el puntero con un reloj (indicando al usuario que ha de esperar), independientemente del cursor empleado para cada control en concreto. Si el valor pasa a cero, se retorna a los cursores habituales.

Por otra parte, cada control recibe eventos del ratón, que podemos manejar desde el programa. Los eventos *MouseDown*, *MouseUp*, *MouseWheel* y *MouseMove*, determinan, respectivamente, si el usuario pulsó un botón, lo levantó, movió la rueda del ratón o movió el ratón de posición.

Dentro de estos eventos, y sólo dentro de ellos, se puede emplear la clase *Mouse*, para determinar qué botón se pulsó (izquierdo, derecho o central) mediante las propiedades *Mouse.Left*, *Mouse.Right* o *Mouse.Middle*, que toman el valor *true* si el botón correspondiente ha sido pulsado o levantado; la posición del ratón dentro del control (*Mouse.X* y *Mouse.Y*); la posición del ratón respecto al escritorio (*Mouse.ScreenX* y *Mouse.ScreenY*); así como el movimiento sobre la rueda del ratón (*Mouse.Delta* y *Mouse.Orientation*).

En la mayor parte de controles, el evento *MouseMove*, se produce sólo si hay, al menos, un botón pulsado del ratón por parte del usuario.

Este pequeño ejemplo permite mover un botón de posición dentro de un formulario, cuando el usuario lo arrastra mientras mantiene pulsado el botón izquierdo.

Creamos un formulario *Form1*, con un botón **Button1**, que incluye este código:

```
PRIVATE pX AS Integer
PRIVATE pY AS Integer

PUBLIC SUB Button1_MouseDown()

  IF Mouse.Left THEN
    '************************************
    ' Almacenamos posiciones iniciales
    '************************************
    pX = Mouse.X
    pY = Mouse.Y
  END IF

END


PUBLIC SUB Button1_MouseMove()

  IF Mouse.Left THEN
    '**************************************
    ' Desplazamos el botón de acuerdo
    ' con la variación de X e Y
    '**************************************
    Button1.Move(Button1.X - px + Mouse.X, Button1.Y -
    py + Mouse.Y)
  END IF

END
```

A diferencia de otros entornos de desarrollo, donde el evento recibe varios parámetros indicando el botón, posición, etc., en Gambas todo ello se conoce a partir de la información almacenada en la clase *Mouse*, que sólo está disponible en estos eventos.

El evento *MouseDown* es cancelable, lo que significa que empleando la instrucción *STOP EVENT* dentro de su código, se evita que se propague y, por tanto, que el control actúe en consecuencia, según su funcionamiento interno habitual (por ejemplo, un botón no lanzaría el evento *Click* si se cancela *MouseDown*).

Otros eventos comunes en los controles son *DblClick* (doble pulsación del ratón), *Menu* (pulsación del botón derecho) o *Click* (pulsación del botón izquierdo). Algunos controles pueden no disponer del evento *Click*.

Teclado

De forma similar al ratón, el teclado se controla con los eventos *KeyPress* y *KeyRelease*. Estos no tienen parámetros.

La clase estática *Key* proporciona la información necesaria para controlar el teclado dentro de estos eventos, de igual modo que la clase *Mouse* dentro de los eventos de ratón.

El evento *KeyPress* es cancelable con la instrucción *STOP EVENT*, de modo que se puede impedir, por ejemplo, que en una caja de texto se impriman determinados caracteres.

En el siguiente ejemplo, se bloquea un **TextBox**, de forma que sólo permita la entrada de números, la pulsación de las teclas **Supr** (borrado) y **BackSpace** (borrado hacia atrás) y el **tabulador** para pasar el foco a otro control.

Para ello llama a **STOP EVENT** cuando el código de la tecla pulsada no es ninguno de los deseados:

```
PUBLIC SUB TextBox1_KeyPress()

  SELECT CASE Key.Code

  CASE 48 TO 57 ' Códigos ASCII de los números
    CASE Key.BackSpace ' Retroceso
    CASE Key.Delete ' Borrado
    CASE Key.Tab ' Tabulador
    CASE ELSE
      STOP EVENT

  END SELECT

END
```


3. 3 Galería de controles

Controles básicos

Tanto *gb.qt* como *gb.gtk* aportan una serie de controles básicos para desarrollar una interfaz gráfica. A continuación se detallan estos controles y sus características principales.

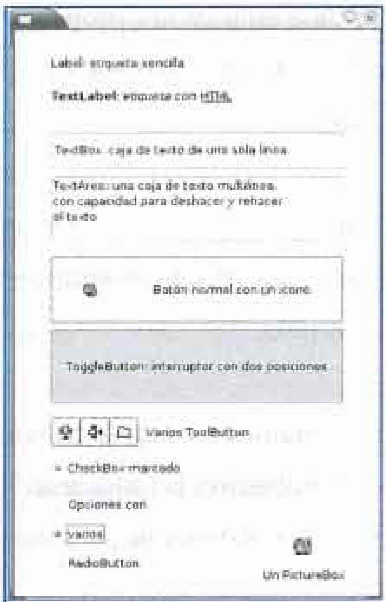


Figura 6. Controles de *gb.qt* y *gb.gtk*.

- **Label:** es una etiqueta simple que contiene una línea de texto de poca longitud. Su única función es mostrar un texto en una posición dentro de un formulario. La propiedad *Text* es la que determina el texto a mostrar en cada momento. Al margen de este uso básico, se puede modificar tanto su color de fondo (*Background*), como el color de primer plano (*ForeGround*). Como el resto de controles,

responde a los eventos de ratón, por lo que en un momento dado, ayudado de *MousePress* y *MouseRelease*, pueden servir para implementar un botón personalizado.

- **TextLabel:** es muy similar al control *Label*, pero tiene la particularidad de que es capaz de mostrar texto formateado en HTML. De esta forma, indicando una cadena con etiquetas HTML en la propiedad *Text*, podremos tener texto que combine negrita, itálica, subrayado y otras características de texto enriquecido.

```
TextLabel1.Text="<b>Texto con HTML</b><br>Dentro de una
<i>etiqueta."
```

El componente *gb.gtk* es capaz de representar menos etiquetas HTML, en las versiones actuales. Por su parte, *gb.qt* permite la inclusión de tablas e imágenes desde un archivo. Un programa desarrollado para ser independiente del toolkit, no debería abusar de esto. Pero un programa expresamente diseñado para trabajar con *gb.qt* puede, por el contrario, sacar provecho de sus características extra.

- **TextBox:** es una caja de texto, de una sola línea, en la cual el usuario puede modificar, copiar, cortar o borrar texto. El texto introducido se recibe o modifica por código mediante la propiedad *Text*. Además, el método *Select* permite seleccionar o resaltar por código una parte del texto, y *Selection* dota de algunas propiedades para conocer el texto que el usuario ha seleccionado.

Gambas emplea codificación UTF-8 en la interfaz gráfica, por lo que un carácter puede suponer 1, 2 o 3 bytes de longitud. Así, escribir *Len(TextBox1.Text)* para conocer la longitud en caracteres de un texto, no siempre dará el resultado esperado. En su lugar, se debe emplear el método *TextBox.Length*, que da siempre la longitud en caracteres del texto.

- **TextArea:** se trata de una caja de texto que es capaz contener múltiples líneas. Se permite también los retornos de carro. Como en el caso del *TextBox*, el método *Selection* y la propiedad *Select* determinan el texto seleccionado. Además, este control dispone de los métodos *Undo* y *Redo* que equivalen a las órdenes *Deshacer* y *Rehacer* de cualquier editor de textos. Es decir, eliminan los últimos cambios del usuario o los vuelven a situar en el texto.

- **Botones**: Gambas tiene tres tipos de botones. El primero, *Button*, es un botón normal, dispone de una propiedad *Text* que indica el texto a mostrar, así como una propiedad *Picture* para mostrar un icono identificativo. Este control dispone del evento *Click* que se dispara cuando se pulsa con el botón izquierdo (o derecho si está configurado para zurdos). Otro tipo de botón es el *ToggleButton*, que mantiene su estado tras una pulsación, es decir, cuando se pulsa una vez, queda presionado, y al pulsarlo otra vez, sale de ese estado. La propiedad *Value* sirve para conocer o variar su estado: *FALSE* significa 'no presionado' y *TRUE* 'presionado'. El siguiente botón, *ToolButton*, es similar, pero sólo muestra un pequeño icono, sin texto. Está diseñado para insertarse en barras de herramientas, habituales en la parte superior de las interfaces, como acceso rápido a ciertas funciones comunes. Puede actuar como un botón normal, si su propiedad *Toggle* vale *FALSE*, o como un interruptor (como un *ToggleButton*), si *Toggle* vale *TRUE*. Así mismo dispone de una propiedad *Border* que de valer *FALSE* dará apariencia plana al botón, y de valer *TRUE* lo mostrará con relieve, como un botón normal.

Los botones *Toggle* pueden ser confusos con algunos temas de KDE o GNOME, siendo difícil para el usuario determinar si se encuentra pulsado o no. Cuando sea posible se empleará un *CheckBox* en lugar de un *ToggleButton*, o bien se resaltará su estado de algún modo, por ejemplo, cambiando el color de fondo o el icono que se muestra. Esto evitará problemas con los futuros usuarios de la aplicación.

- **CheckBox:** muestra un texto determinado por la propiedad *Text*, junto con una caja donde el usuario puede pulsar para marcar o desmarcar la opción.

La propiedad *Value* indica si el usuario ha marcado o no el *CheckBox*, y el evento *Change* informa de cada cambio. Se suele emplear para opciones de configuración que sólo disponen de dos posibles valores: 'Activado o Desactivado', 'Sí o No', 'Verdadero o Falso'.

- **RadioButton**: es similar a *CheckBox*, pero tiene la particularidad de que todos los *RadioButton* existentes, *hijos* de un mismo contenedor, están internamente agrupados, y en cada momento sólo puede haber uno activado. Cuando el usuario activa uno de ellos, el resto se desactiva, por lo que se emplea para seleccionar una opción que excluye a otras dentro de un menú de opciones.

- **PictureBox**: este control tiene la función de mostrar una imagen. Responde a eventos de ratón, por lo que se puede emplear como botón personalizado. Su propiedad *Stretch* permite adaptar la imagen al tamaño del *PictureBox* en cada momento, la propiedad *Border* determina su apariencia plana o con relieve y la propiedad *Picture* representa la imagen a mostrar.

Otros controles básicos misceláneos



Figura 7. Otro tipo de controles.

Otros controles que pueden ayudar a diseñar una interfaz y tienen un propósito mucho más concreto son los que se presentan en la imagen de la izquierda.

- **ProgressBar**: barra de progreso que muestra un porcentaje de forma gráfica. Sirve para dar idea del avance de un proceso que dura mucho tiempo, de forma que el usuario no sienta que la aplicación está colgada mientras trabaja en segundo plano.

- **Slider**: es similar a *ProgressBar* en el sentido de que muestra un porcentaje, pero en este caso el usuario es quien varía su valor. Un buen ejemplo es su uso para subir o bajar el volumen en una aplicación que reproduzca audio. Los valores de volumen se definen entre un rango máximo y mínimo, y el usuario

lo cambia a su gusto. El evento *Change* señala un cambio por parte del usuario en el valor de la escala.

- **MovieBox**: a pesar de su sugerente nombre, no se trata de un reproductor multimedia, sino algo más humilde. Muestra una animación en formato GIF o MNG, sin que el programador deba preocuparse de la sucesión de frames del archivo que se muestra. La propiedad *Path* determina el archivo a reproducir, y *Playing* permite el control de la reproducción, con los valores *TRUE* (reproducir) o *FALSE* (detenido).

- **ScrollBar**: se trata de una barra de scroll para desplazar otro control, habitualmente, de forma que sea el usuario quien determine la posición de éste. La propiedad *Value* indica el valor elegido por el usuario, y el evento *Change* señala cada cambio.

A diferencia de otras interfaces gráficas, donde existen variantes horizontales y verticales para determinados controles, en Gambas los controles *Slider* y *ScrollBar* determinan su orientación automáticamente: si son más anchos que altos serán horizontales, y verticales en caso contrario.

Listas de datos

Existen tres controles diseñados para mostrar listas de diferentes modos:

1. **ListBox**: es una lista simple. Se añaden o eliminan elementos que se representan como una línea de texto cada uno. El usuario tiene capacidad para seleccionarlos o deseleccionarlos. La propiedad *Mode* determina si el usuario no puede seleccionar ninguno, sólo uno o varios.

2. **ListView**: similar a *ListBox*, dispone de capacidades adicionales. Puede representar un icono junto con cada elemento de la lista, y cada uno de ellos está identificado por una clave única de texto, que nos permite hacer búsquedas

de los elementos por su clave. Así mismo dispone de un cursor interno que puede moverse hacia adelante y hacia atrás, lo que lo hace apropiado para interactuar con una aplicación de bases de datos, donde puede representar un campo de una tabla.

3. **ComboBox**: es una lista desplegable. El usuario sólo ve el elemento seleccionado en cada momento y puede desplegar la lista para seleccionar uno u otro.

Otros controles avanzados

A continuación mostramos otros controles más avanzados:

- **TreeView**: sirve para representar elementos en un árbol, de forma que cada nodo puede tener otros nodos *hijos*.

- **ColumnView**: es similar al anterior, pero cada nodo puede disponer de varias columnas.

- **GridView**: sirve como representación de parrilla, de forma que disponemos de registros agrupados en filas y columnas. Es empleado, habitualmente, para interactuar con bases de datos.

3. 4 Diálogos

Gambas aporta una serie de diálogos auxiliares para mostrar o recabar información, interactuando con el usuario.

La clase Message

La clase *Message* se encarga de mostrar una ventana modal al usuario, en la cual podemos definir un texto, que será una información o una pregunta, y una serie de botones para elegir una opción. La clase *Message* es estática, y dispone de una serie de métodos para mostrar distintos tipos de mensajes, que serán reconocibles para el usuario gracias al icono que acompaña a la ventana y que da una idea del carácter del

mensaje. En estos métodos se situará siempre, como primer parámetro, el texto a mostrar y, a continuación, el texto de los botones, hasta un máximo de tres. Si no se indica el texto de los botones, aparecerá sólo un botón indicando OK para que el usuario acepte la lectura del mensaje.

- **Message.Info** (Texto, Boton1): se utiliza para mostrar un mensaje meramente informativo. Sólo permite definir un botón, que normalmente tendrá un texto tal como **OK** o **Aceptar**.



Figura 8. Mensaje informativo.

La clase *Message* también puede ser llamada como una función, de modo que el código:

```
Message.Info ("Mensaje")
```


Es equivalente a:

```
Message("Mensaje")
```

- **Message.Delete** (Texto, Boton1, Boton2, Boton3): se utiliza para indicar que se va a proceder a eliminar algo (archivo, registro de una tabla...), y se solicita al usuario su confirmación.



Figura 9. Mensaje de eliminación.

- **Message.Error** (Texto, Boton1, Boton2, Boton3): se emplea para indicar un mensaje de error.



Figura 10. Mensaje de error.



Figura 11. Mensaje para preguntar.

• **Message.Question** (Texto, Boton1, Boton2, Boton3): es una pregunta al usuario, generalmente para confirmar una acción o una opción de configuración.



Figura 12. Mensaje de advertencia.

• **Message.Warning** (Texto, Boton1, Boton2, Boton3): advierte al usuario de que la acción que va a realizar supone un cierto peligro, por ejemplo, pérdidas de datos de una tabla que podrían ser útiles aún.

Los métodos de la clase *Message* devuelven un número entero que denota el botón que el usuario pulsó. El primer botón comienza en el número 1. Los mensajes son modales, lo que quiere decir que la interacción de la interfaz de usuario con el programa, así como el flujo de éste, quedan bloqueados hasta que se pulse uno de los botones.


```
...
DIM hRes AS INTEGER
hRes=Message.Warning("¿Formatear el disco duro?",
"Sí","No")
IF hRes=1 THEN Formatea_Disco()
...
```



Figura 13. Cuadro de diálogo con botón de cerrar.

Dependiendo del gestor de ventanas del sistema, es posible que los cuadros de diálogo tengan un botón de cerrar. Si el usuario cierra el mensaje de este modo, se devolverá el número del botón existente más alto (en el ejemplo anterior el 2), por tanto la opción menos peligrosa, o que se estima a ejecutar por defecto, se habrá de indicar en el botón más alto.

Los mensajes se han de emplear con prudencia. Una aplicación que continuamente bloquea la interfaz y desvía el foco de atención del usuario, puede llegar a ser realmente molesta. Las interfaces de otros sistemas propietarios se han llegado a hacer famosas en los chistes de las oficinas por las continuas preguntas del estilo '¿Está seguro de que...?'.

La clase Dialog

Esta clase aporta ciertos diálogos comunes, que ahorrarán mucho tiempo de trabajo mecánico al programador de una aplicación gráfica. Dependiendo del componente, lógicamente, tendrán un aspecto similar al resto de aplicaciones GTK+ o QT. Los métodos de esta clase no toman ningún parámetro, y devuelven siempre un valor Bolean, de modo que *TRUE* significa que el usuario ha pulsado el botón **Cancelar**, y *FALSE* significa que aceptó el valor seleccionado. Las posibilidades que nos ofrecen son las siguientes:

- **Dialog.OpenFile()**: sirve para que el usuario seleccione un archivo para su apertura. La propiedad *Path* de la clase se utiliza para situar la ruta inicial en el sistema de archivos. Tras ser mostrado y seleccionado por el usuario, contiene la ruta elegida por éste.

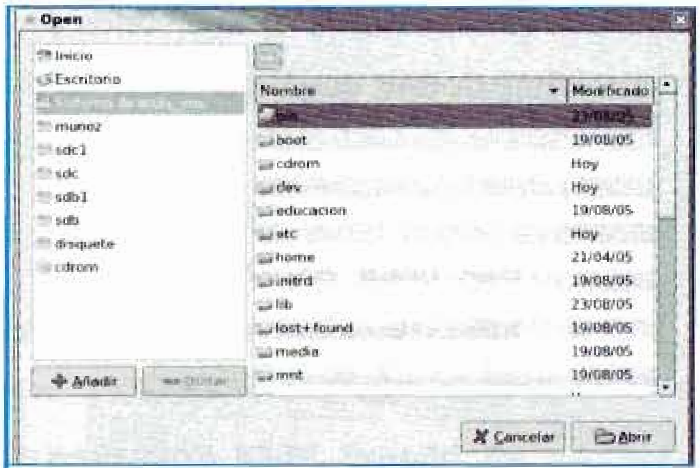


Figura 14. Apertura de archivo.

- **Dialog.SaveFile()**: es similar al anterior, pero el diálogo está orientado a guardar un archivo, de forma que permite elegir uno que no existe. Como en el caso de *OpenFile()*, la propiedad *Path* sirve para situar la ruta inicial y para recabar la ruta seleccionada por el usuario.



Figura 15. Ventana para guardar un archivo.

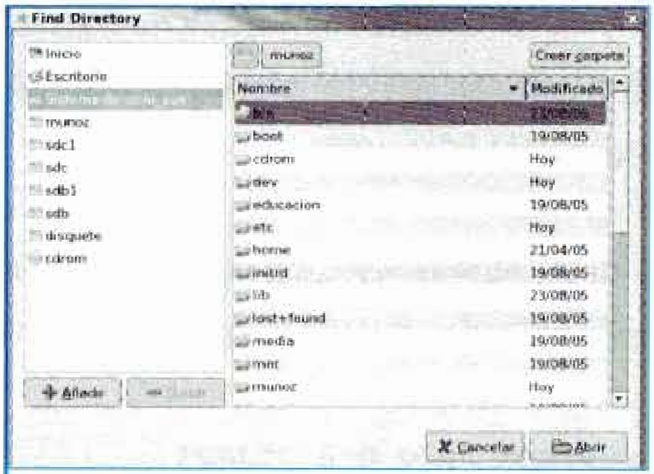


Figura 16. Apertura de carpetas.

- **Dialog.SelectDirectory()**: en este caso, el valor de *Path* no será un archivo, sino una carpeta. El diálogo permite seleccionar sólo carpetas, no archivos.

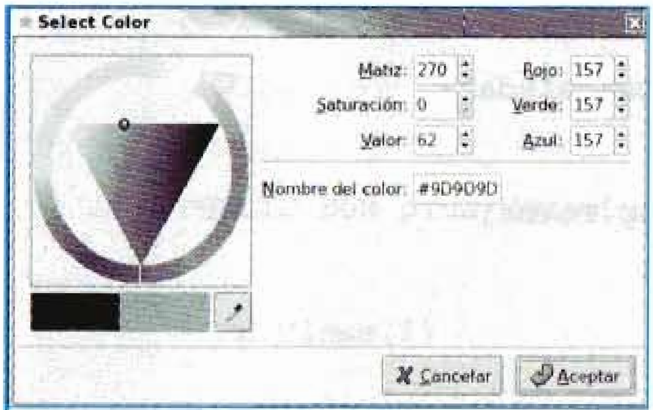


Figura 17. Selección del color.

- **Dialog.SelectColor()**: permite al usuario seleccionar un color personalizado. La propiedad *Color* se emplea de modo similar a la propiedad *Path* de los casos anteriores, es decir, sirve tanto para fijar el color inicial como para determinar el elegido por el usuario.

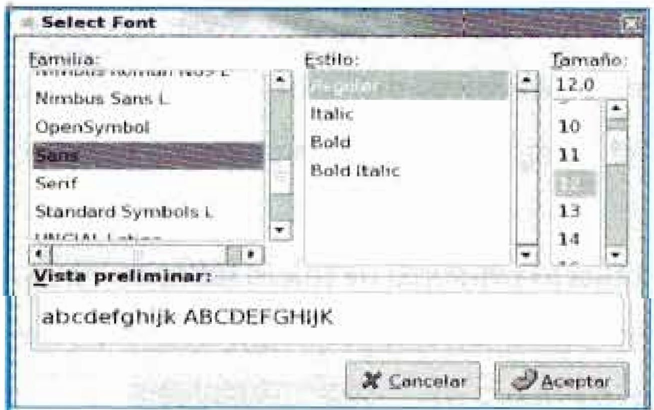


Figura 18. Elección de la fuente.

- **Dialog.SelectFont()**: con él se elige una fuente personalizada. Se trabaja sobre la propiedad *Font* de la clase *Dialog*.

Además de la propiedad *Path* existe una propiedad *Paths* para obtener los valores de los archivos seleccionados, cuando se necesite elegir más de uno a la vez. Para activarla, el método *Dialog.OpenFile* admite un parámetro opcional que habrá de tener valor *TRUE*.

- **Dialog.Filter**: permite indicar filtros para los archivos a mostrar. Se trata de una matriz de cadenas en la que podemos situar, por ejemplo, las extensiones de los archivos a elegir, usando comodines si lo deseamos.

- **Dialog.Title**: permite establecer un título para la ventana, que por defecto corresponderá a la acción a realizar (*Select Font*, *Select Color*, etc.).

```
...
Dialog.Title = "Seleccione imágenes a procesar"
Dialog.Filter = ["*.png", ".jpg"]
IF Dialog.OpenFile(TRUE) THEN
  Message.Info("Acción cancelada")
ELSE
  Procesa_Imagenes(Dialog.Paths)
END IF
...
```


Diálogos personalizados

Al margen de los diálogos ya mencionados, el programador puede crear otros personalizados. Cuando un formulario se muestra de forma *Modal*, es decir, con los métodos *ShowModal()* o *ShowDialog()*, puede devolver un valor entero, que sirve como indicación de la opción elegida por el usuario. Veamos un pequeño ejemplo.

Creamos un programa gráfico, con un formulario principal de inicio llamado **FMain** y otro formulario llamado **FDialogo**. Creamos también tres pequeños iconos en formato *png* o los copiamos de la carpeta */usr/share/pixmaps* del sistema, y los renombramos como *a.png*, *b.png* y *c.png*, de forma que los tengamos disponibles en la carpeta del programa. El formulario principal **FMain** tendrá un *PictureBox* llamado **pImage**, y un botón denominado *btnSelect* con el texto **Icono**. El formulario FDialogo dispondrá de tres controles

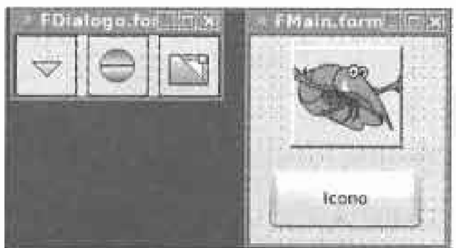


Figura 19. Formularios **FMain** y **FDialogo**.

PictureBox, cada uno de ellos conteniendo uno de los iconos *png* que habíamos situado en el proyecto, y los llamados **pic1**, **pic2** y **pic3**.

Con el código del formulario **FDialogo** lo que conseguiremos es que cada vez que el usuario pulse uno de los controles *PictureBox*, el formulario, que se mostrará de forma modal, devuelva un valor entero que identifquea el icono pulsado.

```
PUBLIC SUB pic1_MouseDown()

  ME.Close(1)

END

PUBLIC SUB pic2_MouseDown()

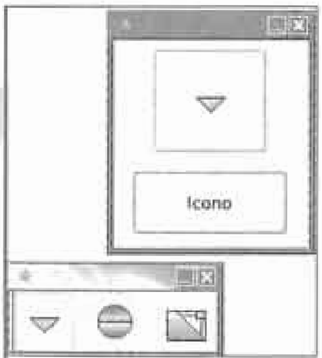
  ME.Close(2)

END

PUBLIC SUB pic3_MouseDown()

  ME.Close(3)

END
```

En cuanto al formulario **FMain**, la pulsación del botón conducirá a mostrar el formulario **FDialogo** de forma modal y, en función del valor devuelto, situará una imagen u otra en el *PictureBox* llamado **pImage**:

Figura 20. Imágenes que hemos situado en el *PictureBox*.

```
PUBLIC SUB BtnSelect_Click()

  SELECT CASE FDialogo.ShowDialog()

    CASE 1
      pImage.Picture = Picture["a.png"]

    CASE 2
      pImage.Picture = Picture["b.png"]

    CASE 3
      pImage.Picture = Picture["c.png"]

  END SELECT

END
```


Tras ejecutarlo, podremos comprobar el resultado. Si el usuario cierra el formulario modal pulsando el aspa del gestor de ventanas, se devolverá el valor por defecto, es decir, cero, lo que equivale a cancelar la selección. Puede plantearse otro problema más complejo: la necesidad de devolver otro tipo de valores, tales como cadenas o referencias a objetos. En este caso, la solución del entero no es válida. Si tratamos de mantener una cadena en una variable del formulario, ésta se liberará tras cerrar el formulario, lo cual no nos sirve. Por ejemplo, modificando el código anterior, de la siguiente manera, para que el formulario **FDialogo** mantenga una cadena con el valor elegido:

```
PUBLIC Valor AS String

PUBLIC SUB pic1_MouseDown()

  Valor = "a.png"
  ME.Close()
```

```
END

PUBLIC SUB pic2_MouseDown()

  Valor = "b.png"
  ME.Close()

END

PUBLIC SUB pic3_MouseDown()

  Valor = "c.png"
  ME.Close()

END
```

Y modificando el formulario principal para que tome el valor de la cadena:


```
PUBLIC SUB BtnSelect_Click()

  FDialogo.ShowDialog()
  IF FDialogo.Valor <> "" THEN
    pImage.Picture = Picture[FDialogo.Valor]
  END IF

END
```

No conseguiremos que funcione, ya que la secuencia es la siguiente:

1. La llamada a **FDialogo.ShowDialog()** crea una instancia de la clase *FDialogo*.

2. Tras la pulsación del usuario, se destruye esa instancia y con ella el valor almacenado en la variable pública *Valor*.

3. Al llamar a **FDialogo.Valor**, se crea otra instancia de *FDialogo*, que tiene la variable *Valor* con su valor por defecto (cadena vacía).

Podemos solventar este problema por dos caminos:

1. El primero consiste en declarar la variable *Valor* como estática. Así la variable no depende de cada instancia, si no de la clase, de forma que no se crea ni se destruye en cada llamada a *FDialogo.ShowModal()*.

2. El segundo consiste en aprovechar la propiedad *Persistent* de los formularios. Poniendo su valor a *TRUE*, un formulario no se destruye cuando el usuario lo cierra pulsando el aspa del gestor de ventanas, ni se llama al método *Close()*, si no que simplemente se oculta. Y si estaba en pantalla de forma modal, el programa abandona este modo y continua su ejecución normal.

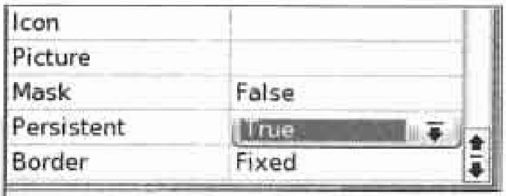


Figura 21. Propiedad *Persistent* con valor *TRUE*.

Así pues, en tiempo de diseño situaremos la propiedad *Persistent* del formulario **FDialogo** a *TRUE*, y modificaremos el código del formulario **FMain** para que destruya el formulario de forma explícita tras haber leído el valor que interesaba.

```
PUBLIC SUB BtnSelect_Click()

  FDialogo.ShowDialog()
  IF FDialogo.Valor <> "" THEN
    pImage.Picture = Picture[FDialogo.Valor]
  END IF

  FDialogo.Delete()

END
```

3. 5 Menús



Figura 22. Localización del **Editor de menú**.

La creación de menús es realmente sencilla ya que un asistente del IDE permite diseñarlos. Tan sólo hay que situarse sobre un formulario, pulsar el botón derecho, y seleccionar la opción **Editor de menu...**

Los menús se crean como árboles, es decir, cada menú de primer nivel, por ejemplo los típicos menús de la barra superior de muchos programas como **Archivo**, **Editar**, **Ver**, **Ayuda**, etc., tendrán menús *hijos* que se sitúan un nivel más profundo que éste y, a su vez, si estos tienen *hijos*, se situarán un nivel más profundo.

Todo ello se controla con los botones con forma de flecha, las verticales permiten cambiar el orden de aparición de los menús, y con las horizontales modificamos la profundidad de estos.

Las propiedades más importantes son el *Nombre*, que es el nombre del objeto menú y que corresponderá con su gestor de eventos, el *Título*, que es el texto que aparecerá en la pantalla, un icono a elegir si lo deseamos, y un posible atajo de teclado para acceder sin necesidad del ratón.



Figura 23. Ejemplo de menús con submenús.

Si el nombre de un menú se deja en blanco, éste se muestra como una barra separadora en lugar de una entrada de menú normal.

En la **Figura 23** podemos ver un ejemplo con un menú principal que tiene tres opciones, y sus correspondientes submenús.

El formulario tendrá el aspecto de la Figura 24. Si en tiempo de diseño pulsamos sobre uno de los menús que no tienen *hijos*, el IDE nos llevará directamente al evento *Click* del menú, que es donde podemos crear el código que se ejecutará cuando el usuario pulse el menú.

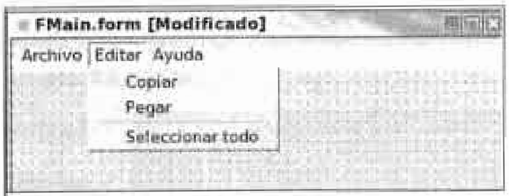


Figura 24. Formulario *FMain*.

```
PUBLIC SUB mnuSeleccionar_Click()

    ...

END
```


Si lo que deseamos es crear un menú de los que se muestran cuando el usuario pulsa, por ejemplo, el botón derecho sobre un formulario u otro control, tendremos que crear un menú de primer nivel con su propiedad visible a *FALSE*, y sus correspondientes *hijos*. Después, al lanzarse el evento del formulario o control que nos interesa, por ejemplo, la pulsación del botón derecho sobre un formulario, que se detecta mediante la gestión del evento *Menu*, indicaremos a nuestro menú invisible que debe mostrarse como un menú contextual:



Figura 25. Menú que vemos al pulsar el botón derecho de un formulario.

```
PUBLIC SUB Form_Menu()

  mnuEditar.Popup()

END
```

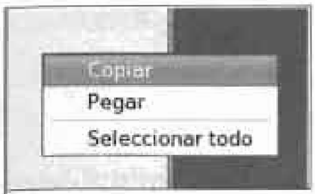


Figura 26. Resultado del menú contextual.

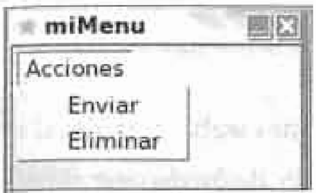


Figura 27. Acciones del menú creado.

Al ejecutar el código, veremos el resultado al pulsar el botón derecho sobre el formulario.

En cuanto a la naturaleza de los menús, no son más que objetos, aunque en este caso no provienen de la clase *Control*, si bien disponen de algunas propiedades comunes como *Text* o *Picture*.

A la hora de crear un menú, habrá de indicarse su objeto *padre*, que podrá ser una ventana o formulario para los menús de primer nivel, u otro menú para los *hijos* del primero. Los menús se pueden crear o destruir también directamente por código:

```
PUBLIC SUB Form_Open()


  DIM h1 AS Menu
  DIM h2 AS Menu
  DIM h3 AS Menu


  h1 = NEW Menu(ME)
  h1.Text = "Acciones"

  h2 = NEW Menu(h1) AS "h2"
  h2.Text = "Enviar"

  h3 = NEW Menu(h1) AS "h3"
  h3.Text = "Eliminar"


END
```

3. 6 Alineación de los controles

Propiedades de alineación

Algunos contenedores disponen de una propiedad *Arrangement* que permite determinar cómo se alinean los controles dentro de un contenedor. Por defecto, el valor de la propiedad es *None*, lo que significa que las posiciones de los controles son libres, como es típico en las interfaces para Windows™ o en algunas librerías como QT.

Sin embargo, en otras librerías gráficas como GTK+, o en las definiciones del lenguaje XUL diseñado por el proyecto Mozilla para aplicaciones web, lo habitual es encontrar contenedores que definen dónde se situará cada *hijo*, de forma que el programador sólo ha de indicar el modo general de alineamiento y los controles se adaptarán en todo momento a dicha alineación.

En principio, diseñar interfaces de esta manera puede ser algo complicado para gente sin experiencia, no obstante, una vez que se toma algo de pericia, aporta grandes ventajas. La principal es que el diseño de ventanas redimensionables con controles variados en su interior es trivial.

Cada usuario puede agrandar o hacer más pequeña la ventana, o variar su relación alto/ancho, y la aplicación seguirá manteniendo un aspecto coherente en cada momento, dentro de unos límites razonables de tamaño.

Gambas define varias posibilidades de alineamiento para los controles:

- **None**: alineamiento libre, el contenedor no decide nada acerca de la posición de sus *hijos*.

- **Horizontal**: todos los controles se alinean de izquierda a derecha, ocupando todo el espacio en vertical dentro del contenedor.

- **Vertical**: todos los controles se alinean de arriba a abajo, ocupando todo el espacio en horizontal dentro del contenedor.

- **LeftRigth**: los controles tratan de alinearse de izquierda a derecha, y si falta espacio, de arriba a abajo.

- **TopBottom:** los controles tratan de alinearse de arriba a abajo, y si falta espacio, de izquierda a derecha.

Además de la propiedad general *Arrangement*, se aporta la propiedad *Padding*, que es un espacio que queda libre en el borde del contenedor, y una propiedad *Spacing*, que determina un espacio de separación entre control y control.

Cada control, por su parte, dispone de la propiedad *Expand*. Si el control se sitúa sobre un contenedor cuya propiedad *Arrangement* es distinta de *None*, el valor *Expand* determina si éste, junto con el resto de controles del contenedor que tengan la propiedad *Expand* a *TRUE*, tratarán de ocupar el espacio libre que quede dentro del contenedor.

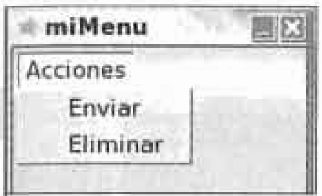


Figura 28. Proyecto **Alineación**.

Para comprobar el efecto de todas estas propiedades, creamos un nuevo proyecto gráfico llamado **Alineacion**, con un sólo formulario de inicio *FMain*, en cuyo interior situaremos dos botones y un *RadioButton*. Tras ejecutarlo veremos el resultado habitual: un formulario con tres controles algo desordenados.

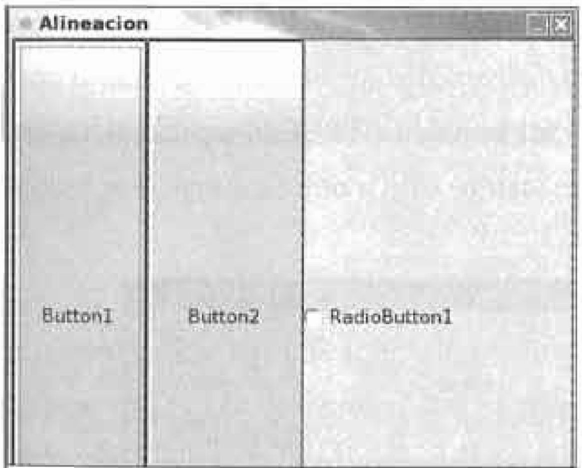


Figura 29. Controles alineados en horizontal.

Cambiamos el valor de la propiedad *Arrangement* a *Horizontal* y ejecutamos el programa.

Ahora los controles se han situado alineados en horizontal, ocupando todo el espacio vertical del contenedor (**Figura 29**).

Ponemos la propiedad *Border* del formulario a *Resizable*, de modo que podamos variar su tamaño. Hacemos varias pruebas de ejecución cambiando su relación altura/anchura.

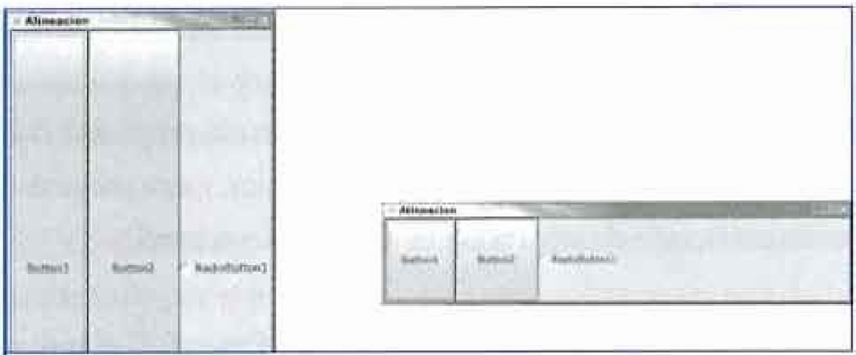


Figura 30. Variación del formulario según altura/anchura.

Como podemos observar, los controles siguen ocupando todo el ancho del contenedor, mientras que el ancho extra queda libre. Situamos ahora la propiedad *Expand* del control **Button2** a *TRUE*, y volvemos a ejecutarlo. Con esta nueva configuración, todo el espacio del contenedor es aprovechado, modificando el ancho del control **Button2**.



Figura 31. Aprovechamiento del espacio del contenedor.

Si situamos la propiedad *Expand* de **Button1** también a *TRUE*, el espacio extra será compartido por ambos controles.



Figura 32. Espacio compartido entre los controladores **Button1** y **Button2**.

Para dejar espacio visible por el borde del contenedor, podemos dar un valor a la propiedad *Padding*, y para separar un poco cada control utilizaremos la propiedad *Spacing*. Los valores indicados son píxeles de separación.

El efecto de la propiedad *Arrangement* no se aprecia en tiempo de diseño, de este modo si por error se cambia a un valor no deseado, podremos revertirlo sin perder las posiciones de cada control dentro del contenedor.

Controles con alineación predefinida

Los controles *Form* (Window), *Panel*, *TabStrip* y *ScrollView* permiten definir las propiedades de alineación, sin embargo, otros controles tienen esta propiedad implícita y no es modificable. Estos son los siguientes:

- **Hbox**: es un panel que siempre tiene alineación horizontal.
- **Vbox**: es un panel con alineación vertical.
- **Hpanel**: sigue el modelo *RightToLeft*.
- **Vpanel**: sigue el modelo *Vpanel*.

Además de estos, los controles *Hsplit* y *Vsplit* son contenedores con un modo de trabajo totalmente distinto: cada control añadido se muestra separado por una barra vertical, en el caso de *Hsplit*, u horizontal, en el caso de *Vsplit*, que el usuario puede mover para agrandar uno u otro control en detrimento del tamaño de su vecino.

Diseño de una aplicación que aprovecha este recurso

A la hora de diseñar una aplicación redimensionable, lo mejor es plantear áreas de trabajo con distinta funcionalidad y agruparlas en distintos paneles horizontales y verticales. Supongamos un clon de los exploradores de archivos habituales. Una zona de trabajo estará formada los típicos botones de menú, que permiten realizar

las tareas más comunes y que presentan un icono y un tooltip explicativo. Otra zona de trabajo puede ser la parte inferior, en la que se muestran datos de estado. Por último, la zona central muestra los archivos y, a su vez, es una zona de trabajo que comprende otras dos: un árbol con las carpetas a la izquierda, y una zona más grande a la derecha con la vista en detalle de los archivos. No crearemos en este ejemplo el código correspondiente, pero sí seguiremos los pasos necesarios para crear la interfaz de un modo adecuado, para conseguir que cada usuario pueda disponer de sus ventanas como mejor lo desee.

En primer lugar, hay tres grandes grupos de trabajo, que van de arriba abajo. Por tanto, lo mejor es definir un formulario con la propiedad *Arrangement* situada a *Vertical*.



Figura 33. Formulario con la propiedad *Arrangement* a *Vertical*.

Dentro de ésta, situaremos tres paneles con alineación horizontal, es decir, tres contenedores *Hbox*. Deseamos que la parte superior con los botones y la inferior con la barra de estado, tengan un ancho fijo, pero la parte central que contiene el cuerpo de la información útil del programa, será redimensionable. Por tanto, la propiedad *Expand* del panel central habrá de tener el valor *TRUE*.

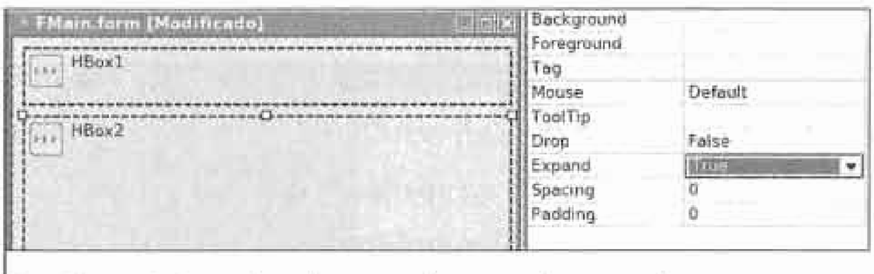


Figura 34. Paneles *Hbox* con alineación horizontal.



Figura 35. Diseño del formulario **FMain**.

La parte superior contendrá diversos botones tipo *ToolButton*, que podemos resaltar con distintos iconos. La parte inferior dispondrá de dos etiquetas *Label* con borde *Sunken*, una de las cuales, según qué parte se quiere hacer extensible, tendrá su propiedad *Expand* a *TRUE*.



Figura 36. Botones auxiliares del formulario.

Dentro del cuerpo se situará una caja vertical a la izquierda (*Vbox*) con otros botones auxiliares. Es el mismo diseño que la zona de botones principal, pero alineada en vertical.



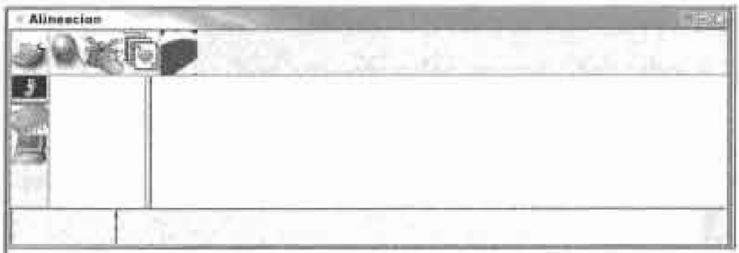
Figura 37. Introducción del control *Hsplit*.

Tras esto, situaremos un contenedor *Hsplit*, de modo que el usuario pueda disponer de una barra para modificar el tamaño relativo del árbol y la zona principal de trabajo. Dentro de él se colocará, a la izquierda, el control *TreeView* para el árbol, así como el control *IconView* para la zona principal.

El control *Hsplit* debe tener la propiedad *Expand* a *TRUE*, para que aproveche todo el espacio libre disponible dentro de su contenedor.

Ya podemos ejecutar el programa para ver la interfaz, añadiendo, si lo deseamos, algo de código para rellenar el árbol y la vista de iconos.

Podemos modificar el alto y ancho de la aplicación, la cual se encargará de mantener, en todo momento, la relación de los distintos controles, sin que sea necesario añadir ningún código de cálculo de posiciones por nuestra parte.



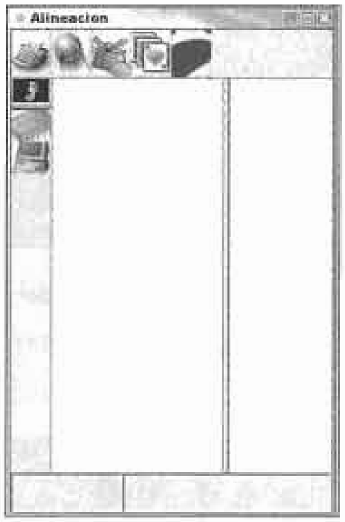


Figura 38. Modificación del alto y ancho de la aplicación **Alineción**.

3. 7 Introducción al dibujo de primitivas

Además de los controles ya diseñados, el programador puede necesitar dibujar gráficos personalizados. La clase estática *Draw* se emplea para dibujar sobre un control, que puede ser un formulario o el control *DrawingArea*.

Dibujar directamente sobre un formulario no tiene demasiado sentido, ya que una vez se refresque la interfaz, por ejemplo tras pasar otra ventana por encima o minimizar y maximizar, se pierde el dibujo realizado sin que tengamos control sobre la situación.

Sin embargo, *DrawingArea* dispone de dos modos de trabajo: en el primero, como en el caso del formulario, se pierde el dibujo una vez se refresca el control, pero el evento *Refresh* nos informa para que redibujemos la parte eliminada; en el segundo, que se activa poniendo la propiedad *Cached* a *TRUE*, el control guarda una caché del dibujo realizado y no genera eventos *Refresh*, sino que redibuja automáticamente la zona clareada.



Figura 39. Proyecto **Dibujando**.

Como ejemplo, crearemos un proyecto gráfico **Dibujando**, que contenga un formulario *FMain*, y en su interior un control *DrawingArea* llamado **Lienzo** y un botón llamado **Dibujar**.

La propiedad *Cached* del control *DrawingArea* se pondrá a *TRUE*.

El código del botón **Dibujar** será el siguiente:

```
PUBLIC SUB Dibujar_Click()

  Draw.Begin(Lienzo)
  Draw.Line(0, 0, Lienzo.W, Lienzo.H)
  Draw.Line(0, Lienzo.H, Lienzo.W, 0)
  Draw.End()

END
```




Figura 40. Resultado de nuestro código.

Al ejecutar el programa y pulsar el botón, aparecerá un aspa creada por nuestro código.

Este código contiene, para cualquier dibujo, el método de trabajo siguiente:

- En primer lugar, se ha de especificar a la clase *Draw* qué control será el empleado para dibujar los elementos que indiquemos, para lo cual se emplea el método **Draw.Begin()** pasando como parámetro el control deseado, en nuestro caso el control *Lienzo*.

- Tras esto, pasamos a dibujar las primitivas que deseamos. Aquí hemos empleado el método **Draw.Line()**, el cual dibuja líneas cuando especificamos los puntos de origen y destino.

- Finalmente, siempre hay que llamar al método **Draw.End()** para que la caché del control *DrawingArea* se dibuje en la pantalla, y se liberen los recursos asociados al proceso de dibujado del control.

La clase *Draw* permite dibujar distintos tipos de primitivas:

- **Draw.Ellipse**: elipses.

- **Draw.Line**: líneas.

- **Draw.Point**: puntos.

- **Draw.Polyline**: varias líneas enlazadas.

- **Draw.Polygon**: polígonos.

- **Draw.Rect**: rectángulos.

En todo momento podemos controlar distintos aspectos del dibujado, utilizando lo siguiente:

- **Draw.BackGround**: color de fondo del pincel.

- **Draw.ForeGround**: color de primer plano del pincel.

- **Draw.FillColor**: color para relleno de elipses o rectángulos.

- **Draw.FillStyle**: utiliza las constantes de la clase *Fill* para determinar el patrón de dibujado (relleno, rayado en horizontal, líneas y puntos, etc.).

Además de las primitivas, podemos dibujar elementos complejos con:

- **Draw.Text**: dibuja un texto en una posición indicada y con la fuente seleccionada por *Draw.Font*.

- **Draw.Image**: dibuja un gráfico almacenado en un objeto *Image*.

- **Draw.Picture**: dibuja un gráfico almacenado en un objeto *Picture*.

Por último, **Draw.Clip** permite seleccionar una área de *Clipping*, es decir, reducir el área útil de dibujo a un rectángulo que especifiquemos, de modo que todo trazo que quede fuera de él se excluirá de ser realmente dibujado.

4. Gestión de procesos

4. Gestión de procesos

4. Gestión de procesos

4. Gestión de procesos

4.1 La ayuda ofrecida por otros programas

Los sistemas GNU/Linux siguen la filosofía de UNIX. Parte de ella consiste en crear pequeños programas especializados en cada tarea, en lugar de generar grandes aplicaciones monolíticas. Como resultado, en GNU/Linux disponemos de muchas pequeñas utilidades de consola capaces de desarrollar casi cualquier tarea que necesitemos. Los programas con interfaz gráfica, habitualmente, son simples *front-ends* para aplicaciones de línea de comandos. Podemos poner como ejemplo el magnífico programa de grabación de CDs y DVDs *K3B*, una aplicación sencilla, bella e intuitiva que, sin embargo, es sólo la portada de aplicaciones como *cdrecord* o *mkisofs*, potentes herramientas de consola. Reproductores de vídeo o audio como *Totem* o *Kmplayer*, recubren también aplicaciones sin interfaz gráfica propia, como el gran *Mplayer*.

Para aquellos que provienen del entorno Win32/VB, puede resultar extraño, acostumbrados a trabajar sólo con los recursos que aporta el propio entorno de programación o añadiendo *llamadas a la API*, esto es, trabajando con librerías del sistema cuando VB se queda corto, pero este modelo de segmentación en pequeñas unidades ofrece pronto grandes ventajas al programador.

GNU/Linux permite desarrollar sin reinventar la rueda. La shell *bash*, común en estos sistemas, junto con las herramientas habituales de consola que acompañan a cualquier distribución, proporcionan todo lo necesario para grabar un CD, reproducir vídeo, gestionar servicios LDAP, transmitir ficheros con ftp, http, smb, scp o nfs, enviar y recibir correos, administrar bases de datos, convertir formatos de ficheros gráficos o de texto y muchas tareas más. No es necesario, en la mayor parte de nuestros desarrollos, entrar en las complejidades de las diversas librerías escritas en C, tratar de solucionar los problemas de conversión de tipos entre un lenguaje de alto nivel y C, ni caer frecuentemente en *violaciones de segmento* por un descuido en un puntero mal gestionado. Basta con consultar la documentación de un comando y llamarlo tal y como haríamos manualmente desde la consola del sistema.

La depuración del programa también resulta más sencilla: basta probar el comando directamente desde un terminal de texto y comprobar los resultados, antes de emberlo en el código Gambas.

4. 2 Gestión potente de procesos

A diferencia de VB, donde lo único que podíamos hacer sin ayuda de la API era lanzar un proceso y desentendernos de él, Gambas permite sincronizar la ejecución de los dos programas, comunicarse con él leyendo y escribiendo por la entrada y salida estándar (*stdin* y *stdout*), conocer el estado (en ejecución o finalizado) y recibir sus mensajes de error por la salida estándar de errores (*stderr*).

4. 3 EXEC

Existen dos comandos para lanzar la ejecución de programas desde Gambas: *EXEC* y *SHELL*. Un programa en ejecución se denomina proceso y a partir de ahora hablaremos de procesos más que de programas. La primera instrucción, EXEC, lanza el comando que indiquemos, acompañado de los parámetros que escribamos:

```
[Variable=] EXEC [Command][ WAIT ][ FOR
(READ|WRITE|READ WRITE) [TO String]
```

Para facilitar la escritura de los parámetros, evitando los problemas que pueden surgir con caracteres especiales tales como los espacios, la sintaxis de *EXEC* indica que hemos de pasar el comando y los parámetros como una matriz o array de cadenas:

```
[Command,param1,param2...]
```


De esta forma, si tenemos que ejecutar, por ejemplo, el comando *cat* con un nombre de archivo tal como *mi archivo.txt*, que tiene un espacio en medio, desaparece la ambigüedad y la necesidad de indicar más caracteres especiales como \, con el fin de eliminar la posibilidad de que el comando tome *mi archivo.txt* como dos parámetros en lugar de uno sólo.

Vamos a ir desgranando poco a poco las diferentes opciones. *Command* es el único parámetro obligatorio y representa el comando a ejecutar. En el modo de trabajo más simple de *EXEC*, éste ejecuta el comando que le indiquemos y se desentiende de él. Ahora, vamos a crear un proyecto de consola con Gambas. Para ello, añadimos un módulo de inicio y, como único código, escribimos:

```
PUBLIC SUB Main()
  EXEC ["ls","-l"]
END
```

Al ejecutarlo se lanza el comando **ls** con el parámetro **-l**, obteniendo un listado de los archivos de la carpeta actual en formato largo. Observemos que nuestra matriz de cadenas está indicada directamente en el comando; podríamos haber hecho también el programa así:

```
PUBLIC SUB Main()

  DIM sCad AS NEW String[]

  sCad.Add("ls")
  sCad.Add("-l")
  EXEC sCad
END
```

Pero el intérprete de Gambas es capaz de reconocer una matriz indicando las cadenas entre corchetes y, de esa forma, nos hemos librado de unas cuantas líneas de código.

Hasta aquí todo lo que hace Gambas es ejecutar el nuevo proceso, pasándole los parámetros, y desentenderse de él. Se ejecuta de forma asíncrona, es decir, el programa Gambas sigue su curso sin esperar a que el proceso hijo finalice. Esto puede ser un inconveniente si tenemos que sincronizar, o esperar a que el proceso acabe, antes de continuar con la siguiente instrucción. Podemos realizar esta tarea de las tres maneras que vemos a continuación.

Palabra clave WAIT

Si añadimos el flag *WAIT* a la instrucción *EXEC*, el programa principal se detendrá hasta que el proceso haya finalizado de forma normal o debido a algún fallo.

Veamos de nuevo el primer ejemplo con pequeñas modificaciones. Vamos a hacer un listado de la carpeta **/dev**, que contiene gran cantidad de archivos. Si los dos procesos se ejecutan de forma asíncrona, obtendremos resultados impredecibles:

```
PUBLIC SUB Main()
  EXEC ["ls","/dev","-l"]
  PRINT "HOLA DESDE GAMBAS"
END
```

A continuación, lo compilamos y ejecutamos varias veces desde un terminal. Observaremos que, como en el listado de más abajo, la frase HOLA DESDE GAMBAS se introduce de forma caprichosa entre el listado generado por el comando ls:

```
...
rwxrwxrwx  1 root  root  5 jun 28 10:55 mouse -> psaux
crw-rw----  1 root  root  13,  32 jun 28 10:55 mouse0
drwxr-xr-x  2 root  root  60 jun 28 10:55 net
crw-rw-rw-  1 root  root  1,  3 jun 28 10:55 null
lrwxrwxrwx  1 root  root  3 jun 28 10:55 par0 -> lp0
HOLA DESDE GAMBAS
crw-rw----  1 daniel usb  99,  0 jun 28 10:55 parport0
crw-rw----  1 root  root  10,  62 jun 28 10:55 pktcdvd
crw-r-----  1 root  root  1,  4 jun 28 10:55 port
crw-------  1 root  root  108,  0 jun 28 10:55 ppp
...
```

Apliquemos ahora el flag WAIT:

```
PUBLIC SUB Main()
  EXEC ["ls","/dev","-l"] WAIT
  PRINT "HOLA DESDE GAMBAS"
END
```

Lo ejecutamos cuantas veces queramos: ya no existe el problema inicial, el programa Gambas espera a que termine de ejecutarse *ls*, y después pasa a la siguiente línea de código. Hemos conseguido sincronizar la ejecución de dos procesos de forma sencilla, simplificando nuestro código, ya que de otro modo tendríamos, por ejemplo,

que haber volcado el listado en un fichero, esperar en un bucle a que el tamaño del fichero dejase de crecer y, a continuación, leerlo y mostrarlo en pantalla.

El descriptor del proceso

Habremos observado que al principio de la sintaxis de *EXEC*, se indica un valor opcional *VARIABLE=*, que recibe algo de retorno tras llamar a *EXEC*. Esta variable es un objeto de la clase *Process*, y lo que se recibe es un descriptor del proceso que hemos lanzado. Los objetos de la clase *Process* tienen una serie de propiedades que nos permiten conocer el estado del proceso, así como actuar sobre él.

Lo que nos interesa ahora es la propiedad *State*, que refleja el estado de ejecución. Cuando se lanza un proceso, el valor de *State* es *Process.Running*, es decir, proceso en ejecución. Si el programa ya ha terminado, la variable *State* podrá tomar dos valores *Process.Stopped*, detenido, finalizado, o *Process.Crashed*, si finalizó debido a un error grave, habitualmente una violación de segmento.

Utilizando esta propiedad tendremos la capacidad de sincronizar los dos procesos de un modo más potente: podremos realizar algunas tareas en nuestro programa Gambas, mientras esperamos a que finalize el proceso auxiliar. Típicamente, lo que haremos será dar algo de feedback o información al usuario de que debe esperar. Como ejemplo, vamos a descargar un fichero desde Internet cuando el usuario pulsa un botón, y a indicarle que estamos trabajando, que no se ha colgado la aplicación, si no que debe esperar con paciencia.

Vamos a trabajar con la aplicación auxiliar *curl*, que es un programa de línea de comandos que permite precisamente lo que queremos: descargar un fichero desde una URL. Si no tenemos *curl* ya instalado, aprovecharemos para hacerlo ahora, ya que está disponible para todas las distribuciones GNU/Linux habituales, así como para FreeBSD. Para ello, consultamos desde Synaptic, Yast, RpmDrake o nuestro gestor habitual de paquetes en otras distribuciones.

Lo que descargaremos es un programa en Gambas llamado *RadioGambas*, cuyo código contiene un buen ejemplo de gestión de procesos y que también podemos estudiar.

Además, nos sirve para escuchar programas de radio emitidos por Internet, lo cual no está mal. La URL es **http://gambas.gnulinex.org/radiogambas/RadioGambas-1.0.1.tar.gz**, aunque podemos consultar en **http://gambas.gnulinex.org/radiogambas** la existencia de versiones más recientes.



Figura 1. Valor **Inactivo** de la etiqueta y botón **Descargar**.

Ahora, vamos a crear un programa gráfico y, en él, un formulario, con un botón llamado **BtnDescarga** y una etiqueta llamada **LblInfo**.

La etiqueta tendrá como propiedad **Text** el valor **Inactivo**, y el botón tendrá el texto **Descargar**.

El código del formulario será el siguiente:

```
PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS STRING


  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", sURL, "-o", User.Home &
  "/RadioGambas.tar.gz"]

  DO WHILE hProc.State = Process.Running
    SELECT CASE LblInfo.Text
      CASE "|"
        LblInfo.Text = "/"
      CASE "/"
        LblInfo.Text = "-"
      CASE "-"
        LblInfo.Text = "\\"
```

```
        CASE "\\"
          LblInfo.Text = "|"
        CASE ELSE
          LblInfo.Text = "|"
      END SELECT
      WAIT 0.1
    LOOP

    LblInfo.Text = "Inactivo"
    Message.Info("Descarga finalizada")

END
```


Ejecutamos el proceso con los parámetros necesarios para que lo descargue en nuestra carpeta personal con el nombre **RadioGambas.tar.gz**, y recibimos un descriptor del proceso. Seguidamente, entramos en un bucle que se ejecuta mientras el proceso está vivo, es decir, mientras el estado del proceso es **Process.Running**. En dicho bucle cambiamos el valor del texto de la etiqueta entre los valores "-", "|", "/" y "\", de forma que se genera la ilusión de un molinillo, con lo que el usuario sabe que algo se está cociendo por debajo. Con la instrucción **WAIT** refrescamos la interfaz. Al terminar, informamos al usuario y ponemos la etiqueta con su valor original.

Probamos a modificar el programa eliminando el código del molinillo y ejecutando el programa de forma síncrona, como explicamos con el flag *WAIT*:

```
hProc=EXEC["curl", sUrl , "-o", System.Home &
"/RadioGambas.tar.gz"] WAIT
```

El funcionamiento es igual de efectivo: el programa descarga el fichero en ambos casos, pero ahora la interfaz de usuario queda bloqueada durante la descarga, lo que puede hacer pensar que nuestro programa se ha colgado y generar algunas llamadas

inútiles a nuestro servicio de atención al cliente, en el peor de los casos. Es importante, por ello, evaluar en qué casos nos conviene usar *WAIT* y en cuáles es mejor informar, de alguna manera, al usuario para que mantenga la calma y las manos lejos del teléfono.

Redirección con TO

En los ejemplos anteriores con el comando *ls*, la salida aparecía directamente en la consola, lo cual no es útil si queremos procesar la información procedente del comando.

Podemos utilizar la palabra clave TO para conseguir dos propósitos de forma sencilla: esperar a que el proceso acabe antes de continuar el programa principal y recibir en una cadena de texto la salida del programa:

```
PUBLIC SUB Main()

  DIM sCads AS NEW String[]
  DIM Buf AS String
  DIM Bucle AS Integer

  EXEC ["ls", "-l"] TO Buf

  sCads = Split(Buf, "\n")
  sCads.Remove(0)
  sCads.Remove(sCads.Count - 1)

  FOR Bucle = 0 TO sCads.Count - 1
    PRINT Left(sCads[Bucle], 10)
  NEXT

END
```

La instrucción EXEC aguarda hasta que finalice el comando **ls**, almacenando en un buffer la salida estándar del comando, que nos devuelve en la cadena **Buf**. A continuación, procesamos la cadena **Buf** separándola en líneas con **Split**, eliminamos la primera y última (sin información útil) y mostramos en pantalla sólo la parte del listado correspondiente a los permisos.

> La salida de los procesos con varias líneas se puede dividir fácilmente empleando la función *Split*, y utilizando como separador el retorno de carro *\n*.

Hasta aquí hemos visto cómo sincronizar los dos procesos, pero aún podemos tener más control sobre él.

Matar un proceso

Además de las propiedades del objeto *Process*, éste ofrece un método de gran importancia: *Kill*, el cual permite matar o acabar con el proceso en cualquier momento. Supongamos que en nuestro programa anterior, la red es desesperadamente lenta y el usuario decide no esperar e interrumpir la descarga. Gracias a *Kill* podemos incluir esa posibilidad en nuestro programa, vamos a ver cómo llevarlo a cabo.

Añadimos al programa anterior un botón llamado **BtnCancelar**, con texto **Cancelar** y con el flag **Enabled** a FALSE, para que inicialmente esté inactivo.

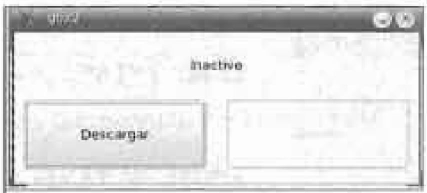


Figura 2. **BtnCancelar Inactivo.**

El nuevo código es el siguiente:

```
PRIVATE hCancel AS BOOLEAN

PUBLIC SUB BtnCancelar_Click()
  hCancel=TRUE
END
```

```
PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS String

  hCancel=FALSE
  BtnCancelar.Enabled=TRUE
  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", sUrl, "-o", User.Home &
  "/RadioGambas.tar.gz"]

  DO WHILE hProc.State = Process.Running
    SELECT CASE LblInfo.Text
      CASE "|"
        LblInfo.Text = "/"
      CASE "/"
        LblInfo.Text = "-"
      CASE "-"
        LblInfo.Text = "\\"
      CASE "\\"
        LblInfo.Text = "|"
      CASE ELSE
        LblInfo.Text = "|"
    END SELECT
    WAIT 0.1
    IF hCancel=TRUE THEN
      hProc.Kill()
      Message.Warning("Proceso cancelado")
      LblInfo.Text = "Inactivo"
      BtnCancelar.Enabled=FALSE
      RETURN
    END IF
```

```
        LOOP

        LblInfo.Text = "Inactivo"
        Message.Info("Descarga finalizada")

      END
```

Disponemos de una variable global, **hCancel**, para saber si el usuario ha pulsado el botón **Cancelar**. Al iniciar la descarga, ponemos esta variable a **FALSE**. Si el usuario pulsa el botón **BtnCancelar**, dicha variable toma el valor **TRUE**. En nuestro bucle, comprobamos el valor de **hCancel** y, de ser **TRUE**, matamos el proceso con **Kill()**, devolvemos la interfaz al estado inactivo, informamos al usuario y salimos de la subrutina.

Aunque ya tenemos sincronización del proceso y podemos acabar con él a nuestro antojo, el feedback que hasta aquí hemos proporcionado al usuario es algo pobre: el molinillo no nos sirve para conocer cuál es el estado real de la descarga, ni hacernos idea de cuánto más habremos de esperar.

Sin embargo, *curl* está emitiendo un informe por la salida estándar de errores *stderr* (o texto impreso por la consola, si no estamos familiarizados con ese término), que podemos aprovechar.

□□□□□ Redirección de la salida estándar de errores

Si estamos familiarizados con el lenguaje C, los mensajes se envían a la salida estándar con la función *printf()* y a la salida estándar de errores mediante *perror()*, ambas incluidas en la librería estándar de C (*glibc* en sistemas GNU/Linux).

Los procesos pueden enviar texto a la consola por dos vías, la primera es utilizando la salida estándar y, la segunda, la salida estándar de errores. Los dos caminos separados se utilizan para diferenciar qué tipos de mensajes se envían. Por la salida

estándar (*stdout*) se suele emitir información útil. Como veremos en el siguiente apartado, el programa *curl* emite el fichero recibido por la salida estándar salvo que indiquemos expresamente el fichero dónde depositarla. Por la salida estándar de errores (*stderr*) se emiten mensajes de estado, de advertencia o de error. *Curl* aprovecha esta salida para indicar el estado de la descarga o para informar de un error en el intento de conexión.

Centrándonos más en *curl*, si aplicamos el parámetro -#, el programa va mostrando una barra de progreso formada por el símbolo # (almohadilla o sostenido) y un indicador del tanto por ciento descargado.

Gambas permite recoger el contenido tanto de la salida estándar como de la salida estándar de errores, si aplicamos el flag *FOR READ* a la hora de ejecutarlo.

Cuando el proceso hijo envía una cadena por la salida estándar de errores (llamando a *perror()*, si está escrito en C), nuestro programa Gambas recibirá un evento Error() procedente de la clase **Process**. La sintaxis de este evento es:

```
PUBLIC SUB Process_Error(sError As String)
  ...
END
```

En la cadena **sError** recibiremos el mensaje de error, o informativo, procedente del proceso hijo. Vamos entonces a modificar el programa para recibir el porcentaje de descarga y representarlo en la etiqueta, en lugar del molinillo.

```
PRIVATE hCancel AS Boolean

PUBLIC SUB BtnCancelar_Click()
  hCancel = TRUE
END
```

```
PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process
  DIM sUrl AS String

  hCancel = FALSE
  BtnCancelar.Enabled = TRUE
  sUrl="http://gambas.gnulinex.org/radiogambas/
  RadioGambas-1.0.1.tar.gz"
  hProc = EXEC ["curl", , "-o", User.Home &
  "/RadioGambas.tar.gz", "-#"] FOR READ

  DO WHILE hProc.State = Process.Running
    WAIT 0.1
    IF hCancel = TRUE THEN
      hProc.Kill()
      Message.Warning("Proceso cancelado")
      LblInfo.Text = "Inactivo"
      BtnCancelar.Enabled = FALSE
      RETURN
    END IF

  LOOP
```

```
        LblInfo.Text = "Inactivo"
        Message.Info("Descarga finalizada")

END
```

En esta ocasión ejecutamos el programa **curl** con el parámetro adicional -#. Cada vez que nuestro programa recibe un evento **Error de Process**, éste se trata en nuestro código: tomamos la cadena, eliminamos con **Trim()** los posibles caracteres especiales de control que *curl* usa para mantener el cursor siempre en la misma línea, de modo que dé la impresión que la barra de progreso avanza (se trata de los llamados caracteres ANSI de control), separamos la cadena en varias subcadenas tomando el carácter de espacio como base y nos quedamos con la última subcadena, que es la que contiene el dato del porcentaje, para representarla en la etiqueta.

□□□□□ Redirección de la salida estándar

Supongamos ahora que no deseamos guardar un fichero, sino mostrar directamente su contenido en pantalla. Por ejemplo, en la URL **http://www.gnu.org/licenses/gpl.txt**, disponemos de un fichero que, en texto plano, contiene la licencia GPL original en inglés.

La primera opción sería guardar el fichero y, una vez finalizado el proceso, leerlo y representarlo en pantalla, pero de este modo complicamos el código ya que tenemos que crear un fichero en alguna ubicación (por ejemplo, */tmp*), leerlo y luego borrarlo para no dejar restos en el disco duro. Como indicamos en el anterior apartado, el contenido de la salida estándar también puede ser leído y *curl* envía el fichero a la salida estándar salvo que, como en nuestros ejemplos anteriores, especifiquemos un fichero donde depositar los datos recibidos. La sintaxis del evento generado por la recepción de datos procedentes de la salida estándar del proceso, es algo diferente de la de los erorres o informativos que vimos antes:

```
PUBLIC SUB Process_Read()
  ...
END
```

En este caso no hay ninguna cadena para recibir la información, por el contrario, cada objeto **Process** se comporta como un flujo o *stream*, lo que en otras palabras significa que podemos trabajar con él del mismo modo que haríamos con ficheros abiertos con *OPEN*, pudiendo utilizar, por tanto, **READ** o **LINE INPUT**.

Hay que recordar también que Gambas provee una palabra clave, *LAST*, que dentro del evento representa de forma genérica al objeto que lo generó. Podemos, entonces, leer el contenido de la salida estándar del proceso utilizando *LAST* como parámetro de las instrucciones relacionadas con lectura y escritura de procesos. Por ejemplo, para leer una línea completa, haríamos:

```
PUBLIC SUB Process_Read()

  DIM sCad AS STRING

  LINE INPUT #LAST,sCad
  PRINT sCad

END
```


Para nuestro programa, añadiremos una caja de texto (*TextArea*), llamándola **TxtLicencia**, con su texto inicial en blanco y en ella pondremos el contenido del fichero **gpl.txt**, conforme lo recibimos.



Figura 3. Caja de texto **TxtLicencia**.

El código es el siguiente:

```
PRIVATE hCancel AS Boolean

PUBLIC SUB BtnCancelar_Click()
  hCancel = TRUE
END
```

```
PUBLIC SUB Process_Read()

  DIM sCad AS String

  LINE INPUT #LAST, sCad
  TxTLicencia.Text = TxTLicencia.Text & sCad & "\n"

END

PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB BtnDescarga_Click()

  DIM hProc AS Process

  TxtLicencia.Text = ""
  hCancel = FALSE
  BtnCancelar.Enabled = TRUE
  hProc = EXEC ["curl", "http://www.gnu.org/licenses/
  gpl.txt", "-#"] FOR READ

  DO WHILE hProc.State = Process.Running
    WAIT 0.1
    IF hCancel = TRUE THEN
      hProc.Kill()
```

```
      Message.Warning("Proceso cancelado")
      LblInfo.Text = "Inactivo"
      BtnCancelar.Enabled = FALSE
      RETURN
    END IF

  LOOP

  LblInfo.Text = "Inactivo"
  Message.Info("Descarga finalizada")

END
```

En el evento Read() leemos una línea y la añadimos al contenido previo de TxtLicencia, más un retorno de carro para separar cada línea.

□□□□□ Evento Kill() y la propiedad Value

Además del método *Kill()*, la clase *Process* emite un evento **Kill()** cuando un proceso ha finalizado, sea de forma normal o por un error. La sintaxis de dicho evento es:

```
PUBLIC SUB Kill()
  ...
END
```

Hasta ahora, en nuestro gestor de descarga hemos esperado en un bucle a que el proceso acabe, pero podemos crear una estructura más elegante, más adaptada a la programación orientada a objetos, valiéndonos de dicho evento: en lugar de esperar en un bucle, aguardaremos tranquilamente y sin hacer absolutamente nada en nuestro programa principal hasta recibir el evento **Kill()**, bien sea porque la descarga ha finalizado, bien sea porque el usuario ha decidido cancelar el proceso:

```
PRIVATE hCancel AS Boolean
PRIVATE hProc AS Process
```

```
PUBLIC SUB BtnCancelar_Click()

  hCancel = TRUE
  hProc.Kill()

END

PUBLIC SUB Process_Error(Err AS String)

  DIM sCad AS String[]

  Err = Trim(Err)
  sCad = Split(Err, " ")
  LblInfo.Text = sCad[sCad.Count - 1]

END

PUBLIC SUB Process_Kill()

  IF hCancel = TRUE THEN
    Message.Warning("Proceso cancelado")
  ELSE
    Message.Info("Descarga finalizada")
  END IF

  LblInfo.Text = "Inactivo"
  BtnCancelar.Enabled = FALSE

END

PUBLIC SUB BtnDescarga_Click()

  DIM sCad AS String
```

```
        TxtLicencia.Text = ""
        hCancel = FALSE
        BtnCancelar.Enabled = TRUE
        hProc = EXEC ["curl", "http://www.gnu.org/licenses/
        gpl.txt", "-#"] FOR READ

END
```

Hemos cambiado la declaración de **hProc** de la función **BtnDescarga_Click()** al inicio de nuestro formulario, para que sea accesible también desde el evento **BtnCancelar_Click()**. Ahora lanzamos el proceso y ya no esperamos en un bucle. Si el usuario decide cancelar la descarga, matamos el proceso en el mismo evento **BtnCancelar_Click()**.

Tanto si el programa termina normalmente como debido a una cancelación, aprovechamos el evento **Process_Kill()** para informar al usuario y devolver la interfaz a su estado inicial (con el botón **BtnCancelar** deshabilitado, y la etiqueta marcando **Inactivo**).

El resultado final: menos gasto de recursos (el programa principal no ha de ejecutar el bucle constantemente), menos líneas de código y mejor estructuración de todo el proceso.

Por otro lado, la mayor parte de los programas de consola (y gráficos) devuelven al sistema un código de error cuando finalizan. La costumbre es que se devuelva un cero si todo fue bien y otro valor cuando algo falló.

Desde la consola podemos hacer una prueba ejecutando en una ventana de terminal estas dos órdenes consecutivas:

```
$ ls -l /dev
$ echo $?
```

Al ejecutar **echo $?** obtendremos un cero. La shell del sistema almacenó el último código de error disponible en la variable **$?** y, a continuación, lo hemos mostrado en pantalla: todo fue bien. Ahora ejecutamos la siguiente orden:

```
$ ls -l /fichero/que/no/existe
$ echo $?
```

En esta ocasión obtendremos un seis, un código de error que en el caso del comando **ls** implica que la ruta no existía (siempre y cuando no haya, por alguna extraña razón que desconozcamos, un fichero en nuestro sistema cuya ruta sea **/fichero/que/no/existe**).

Si *curl* no puede acceder a la URL, por cualquier circunstancia, devolverá un valor distinto de 0. Nosotros podemos leer en nuestro código ese valor, mediante la propiedad *Value*, e informar al usuario del error. A continuación, modificamos el código del programa anterior (página 142 en adelante), de forma que el evento *Kill* quede así:


```
PUBLIC SUB Process_Kill()

  IF LAST.Value<>0 THEN
    Message.Error ("Error en la descarga")
  ELSE
     IF hCancel = TRUE THEN
       Message.Warning("Proceso cancelado")
     ELSE
       Message.Info("Descarga finalizada")
     END IF
  END IF

  LblInfo.Text = "Inactivo"
  BtnCancelar.Enabled = FALSE

END
```

Probemos el programa desconectando el módem de Internet, desenchufando el cable de red o deshabilitando la red de nuestro sistema. Observaremos que ahora podemos determinar el éxito o fallo de la descarga, además de su cancelación.

Redireccion de la entrada estandar, el uso de CLOSE

Los programas, además de emitir información a través de *stdin* y *stderr*, la pueden recibir a través de la consola, según el usuario teclea órdenes. Esta recepción se realiza mediante la entrada estándar o *stdin*, y si tenemos conocimientos de C sabremos que podemos utilizarla con funciones como *scanf()* o *getchar()*. Con Gambas, podemos emplear el flag *FOR WRITE* para indicar al intérprete que estamos interesados en escribir datos para el proceso hijo. Una vez lanzado el proceso de este modo, podemos usar las funciones normales de escritura de archivos con el descriptor del proceso (*PRINT*, *WRITE*).

Supongamos un programa con un área de texto (*TextArea*) llamada **TxtTexto**, en la que escribimos cualquier cosa, y deseamos conocer el número de líneas (separadas con retornos de carro), que hemos escrito. Podemos, para ello, usar el comando wc (significa *Word Counter* y no otra cosa), con el parámetro -l (número de líneas). En nuestro ejemplo, llamaremos al programa, escribiremos el contenido de **TxtTexto** al proceso, y recibiremos el resultado por la entrada estándar. Puesto que vamos a leer y escribir, podemos combinar los flags **READ** y **WRITE**. Para ello, creamos un proyecto gráfico con un botón **BtnContar**, con el texto **Contar**, y un *TextArea* llamado **TxtTexto**, con el texto en blanco inicialmente para que después escribamos un texto.

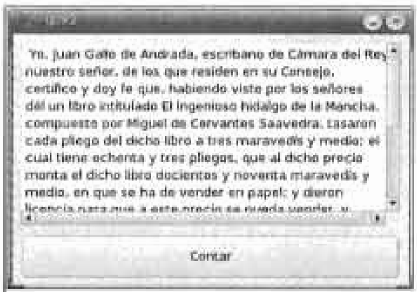


Figura 4. Proyecto con botón **Contar** y área de texto **TxtTexto**.

```
PRIVATE hProc AS Process

PUBLIC SUB Process_Read()

  DIM sCad AS String
```

```
    LINE INPUT #hProc, sCad
    Message("El texto tiene : " & sCad & " líneas")

END

PUBLIC SUB BtnContar_Click()

    hProc = EXEC ["wc", "-l"] FOR READ WRITE
    PRINT #hProc, TxtTexto.Text
    CLOSE #hProc

END
```

n la función BtnContar_Click(), ejecutamos el programa indicando al intérprete ue deseamos acceso de lectura y escritura al proceso, escribimos en éste el contenido de la caja de texto y, a continuación, ejecutamos CLOSE sobre el proceso.

eguidamente, ejecutamos wc -l desde una terminal de línea de comandos. bservaremos que podemos ir escribiendo en el proceso todo lo que queramos. Así, cleamos varias líneas y después pulsamos a la vez las teclas **Control** + D.

s entonces cuando nos devuelve el número de líneas que hemos escrito. Al manar el carácter especial CTRL+D, estamos indicando al proceso que hemos cerra- el flujo de datos y, por tanto, procede que él compute lo recibido y devuelva el sultado.

uando en el programa Gambas ejecutamos CLOSE sobre un descriptor de fiche-, el resultado es similar: se cierra la redirección entre la salida estándar del proce- hijo y nuestro proceso principal, con lo cual el primero queda informado de que recibido todos los datos.

resto del programa es trivial: en el evento READ recibimos la cadena que contie- el número de líneas y lo mostramos al usuario en un mensaje.

Notas finales sobre el objeto Process

El objeto *Process* tiene un método *Signal()* usado por el propio depurador de Gambas. Permite enviar señales al proceso en curso, pero su uso no es aconsejado, ya que interfiere con el resto del código del intérprete.

El objeto dispone también de una propiedad *Id*, que es un handle o descriptor del archivo. Si conocemos C o C++, se trata del identificador de proceso o PID del programa hijo que se obtiene tras una llamada a *fork()*, que es el modo de crear nuevos procesos en sistemas UNIX. Este valor puede servir para buscar el proceso, por ejemplo, ejecutando *ps -le*, variar su propiedad con *nice*, o utilizarlo junto con posteriores llamadas a funciones de C (veremos en otro capítulo cómo hacerlo desde Gambas) para controlar dicho proceso.

Los eventos generados al trabajar con procesos son estáticos. Habremos observado que en los sucesivos ejemplos se ha indicado **Process_Read** y **Process_Kill**. Si tenemos varios procesos en ejecución, siempre podemos diferenciar qué proceso ha generado el evento mediante la palabra clave LAST y actuar en consecuencia:


```
PRIVATE hProc1 AS Process
PRIVATE hProc2 AS Process
...
PUBLIC SUB Process_Read()
  IF LAST=hProc1 THEN ...

END
```

4. 4 SHELL

SHELL es similar a *EXEC*, pero en este caso lanza la shell o intérprete de comandos del sistema, habitualmente *BASH*, en las distribuciones GNU/Linux y le pasa el comando que le indiquemos. Esto permite, por ejemplo, dar órdenes al sistema tales como *export* o *cd*, que son parte de *BASH* y no comandos independientes.

También da vía libre al uso de tuberías | y operadores de redirección como > o *2>* entre otras características de la shell. La sintaxis de SHELL es la siguiente:

```
[Variable = ] SHELL Command [ WAIT ] [ FOR ( READ |
WRITE | READ WRITE )
```

Es muy similar a la de *EXEC*, pero en este caso **Command** es una cadena de texto y no una matriz de cadenas, ya que este valor se pasa directamente a la shell del sistema que, dependiendo de sus características, interpretará los espacios y otros símbolos especiales de un modo u otro. Queda pues, por tanto, como trabajo para el programador, añadir el formateo adecuado para que la shell interprete la orden correctamente.

5. Gestión de bases de datos

5. Gestión de bases de datos
5. Gestión de bases de datos
5. Gestión de bases de datos

5.1 Sistemas de bases de datos

A grandes rasgos, un sistema de bases de datos es una aplicación que permite almacenar y consultar información de forma sencilla. Casi todas las bases de datos actuales posibilitan la interacción con el usuario o programador, a través de un lenguaje llamado SQL. Es importante, antes de proceder a trabajar con una base de datos, conocer este lenguaje, mediante el cual se consultan y modifican datos, además de permitir crear la estructura de la misma (tablas, campos, etc.).

Queda fuera del alcance de este libro explicar el lenguaje SQL, si bien se indican algunas nociones básicas a lo largo de los ejemplos expuestos.

Existen muchas bases de datos distintas, pero podemos clasificarlas en dos tipos. En el primero, existe un programa llamado *Servidor de Bases de Datos*, que gestiona la información almacenada. El servidor mantiene distintas bases de datos, controla los permisos de acceso de cada usuario y permite varias conexiones simultáneas desde distintos equipos cliente. La estructura interna de las bases es tarea del servidor, no del programador de la aplicación. Es decir, éste que no ha de preocuparse de aspectos tales como la ubicación de los ficheros que contienen cada base o tabla. Un ejemplo de este modelo es MySQL (**http://www.mysql.org**).

En el segundo tipo, la base de datos no es más que un archivo alojado en nuestro disco duro. El programador ha de ubicar la base, mantener las copias de seguridad y realizar el mantenimiento que proceda. Normalmente son bases de datos locales, accesibles sólo desde el propio equipo, y habitualmente no permiten más de una conexión de forma simultánea, al menos en modo lectura/escritura. A cambio, consumen muchos menos recursos en el PC, no requieren la instalación y administración de un servidor de bases de datos y suelen ser más fáciles de transportar (por ejemplo, pueden ser grabadas en un CD). En GNU/Linux, la base de datos más popular de este tipo, tal vez, es Sqlite (**http://www.sqlite.org**).

A la hora de diseñar nuestra aplicación, hemos de tener presentes dos aspectos fundamentales: el volumen de datos a manejar y la sencillez de administración requerida. Para una aplicación pequeña, como puede ser una agenda personal, una base de datos de CDs y DVDs, la información relativa a los alumnos de una clase o la gestión de un pequeño comercio, una base de datos Sqlite conjuga la potencia suficiente con una instalación sencilla. El cliente puede recibir el programa e instalarlo con poca ayuda adicional, la base se puede crear en el primer arranque sin configuración previa y el usuario puede hacer sus copias de seguridad periódicamente o llevársela a casa en un CD o en una llave USB para seguir trabajando.

Para la gestión, por ejemplo, de un comercio grande, una aplicación profesional de gestión de nóminas, la consulta de alumnos e inscripciones en un instituto, o la recopilación de información de pacientes de un hospital, una base como Sqlite es totalmente inadecuada, tanto por el volumen de datos almacenados, como por

la necesidad de la consulta e ingreso de datos desde diversos puestos de trabajo de forma simultánea. Aquí tendremos que estudiar qué servidor de bases de datos es el más adecuado para nuestro sistema. En el entorno de software libre, las dos opciones más probadas y de mayor prestigio son MySQL y PostgreSQL (**http://www.postgresql.org/**). Ambos sistemas poseen, además, versiones con soporte comercial.

5. 2 Bases de datos y Gambas

Gambas tiene estructurado el acceso a bases de datos mediante drivers. Estos son módulos de código escritos por diversos programadores específicamente para comunicarse con una base de datos determinada, lo que permite acceder a distintas bases utilizando el mismo código. Como veremos más adelante, basta con especificar el tipo de bases de datos a utilizar, y el resto del código funcionará, posiblemente, sin modificaciones, con independencia de la base de datos utilizada.

Gambas puede manejar varios tipos de bases de datos. Dispone, en este momento, de tres drivers específicos: Sqlite, MySQL y Postgres. Además cuenta con un driver ODBC, el cual es un estándar para comunicar aplicaciones con bases de datos. Por lo tanto, podemos acceder con Gambas a cualquier base que soporte dicho estándar. Esto permite entrar, por ejemplo, a bases de MS SQL Server o Firebird.

A la hora de elegir un driver u otro, tendremos presente que los drivers específicos están optimizados y ofrecen una mayor velocidad de transferencia de datos. Sólo cuando no dispongamos de uno específico, usaremos el ODBC.

Adentrándonos en la estructura de Gambas para bases de datos, cualquier aplicación que use esta característica, necesitará el componente *gb.db* como dependencia. Los drivers para cada sistema de bases de datos son también componentes, pero el programador no ha de marcarlos como dependencias. Una vez que indiquemos a qué sistema nos vamos a conectar, el intérprete de Gambas tratará de cargar el driver específico.

Estos componentes especiales son:

- *gb.db.sqlite*: Sqlite versión 2 o anterior.
- *gb.db.sqlite3*: Sqlite versión 3 o posterior.
- *gb.db.mysql*: MySQL.
- *gb.db.postgres*: PostgreSQL.
- *gb.db.odbc*: genérico ODBC.

5. 3 Gambas-database-manager, el gestor gráfico

Antes de explicar el modelo de programación para bases de datos de Gambas, vamos a aprender a utilizar el Gestor de Bases de Datos, que es una aplicación escrita en este programa, la cual dota al entorno gráfico de desarrollo de una herramienta para administrar, de forma sencilla, múltiples bases de datos.

El Gestor de Bases de Datos escrito en Gambas, es un programa extenso y es Software Libre. Por tanto, el estudio de su código nos puede ayudar a resolver dudas puntuales que surjan a la hora de usar el componente *gb.db*.

Crear una base

Vamos a aprender a crear nuestra propia base. Para ello desarrollamos un proyecto nuevo, o bien abrimos cualquiera que ya tengamos en nuestro equipo. Acudimos al menú **Herramientas** y seleccionamos el **Gestor de Bases de Datos.** También podemos ejecutarlo directamente desde la consola, con el comando **gambas-database-manager.**

Tras pulsar la opción correspondiente, nos preguntará una contraseña. Dicha contraseña se emplea para almacenar encriptados los datos de usuarios y contraseñas que

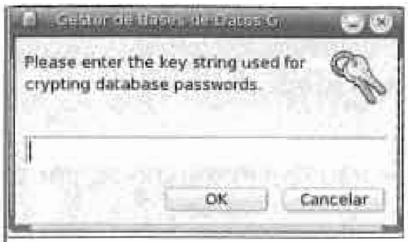


Figura 1. Introducción de contraseña.

deseemos gestionar desde este programa, para evitar que otro usuario pueda leerlos con facilidad consultando los ficheros del entorno Gambas. La contraseña que introduzcamos habrá de tener 8 caracteres como mínimo, y se preguntará cada vez que arranquemos el programa, para desencriptar contraseñas ya almacenadas en sesiones previas.

Tras este paso, aparece el gestor en sí, que en principio está vacío, ya que no hemos configurado ninguna conexión.

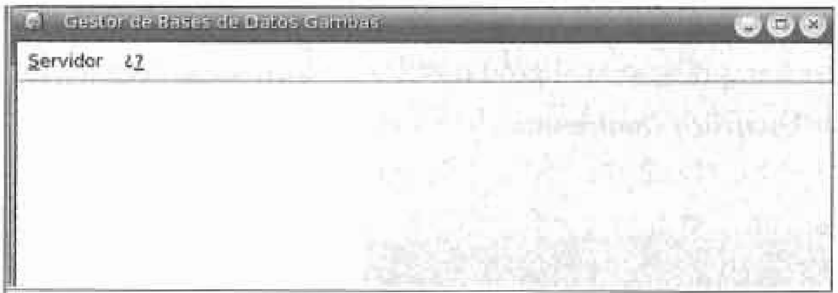


Figura 2. Gestor de bases de datos de Gambas.

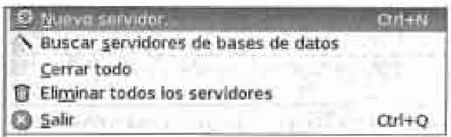


Figura 3. Opción **Nuevo servidor**.

Pulsamos en el menú **Servidor** y elegimos la opción **Nuevo servidor...** Hemos de especificar, a continuación, los datos relativos a la conexión que deseamos establecer (**Figura 4**).

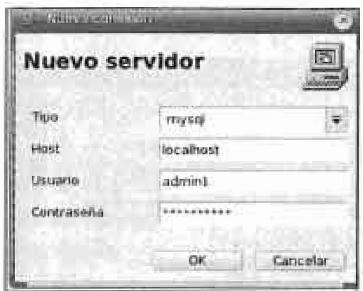


Figura 4. Datos de la conexión.

El primer dato, **Tipo**, se refiere al driver que emplearemos: *sqlite*, *sqlite3*, *mysql*, *postgres* u *odbc*. **Host** es el nombre del equipo o dirección IP donde reside el servidor de bases de datos. Como excepción, para las conexiones *sqlite* el Host no será otra cosa que la carpeta donde se encuentran alojadas las bases de datos, es decir, una ruta absoluta dentro del sistema de archivos, como puede ser */home/usuario/bases*.

El tercer y cuarto dato son el nombre de **Usuario** y la **Contraseña** para acceder al sistema de bases de datos, que determinan los distintos privilegios del usuario.

En el caso excepcional de Sqlite, las bases de datos son simplemente archivos, por lo que los permisos vendrán definidos por los privilegios de un usuario y su grupo en el sistema de archivos (lectura y/o escritura). No procede en ese caso, por tanto, especificar los datos de usuario/contraseña.

Supongamos dos escenarios básicos. En el primero trabajaremos sobre un servidor MySQL en nuestro equipo, con un usuario llamado *admin1* y una contraseña para dicho usuario. En **Tipo** indicaremos el driver *mysql*; como **Host**, al tratarse del propio equipo, se puede indicar o bien *localhost* o *127.0.0.1*, que es una dirección IP que siempre apunta al propio PC. Introduciremos, después, los datos de nombre de **Usuario** y **Contraseña**.

En cuanto a la instalación y administración de MySQL, es muy recomendable consultar la documentación disponible en la propia página de esta base de datos: **http://dev.mysql.com/doc/**.

En el segundo escenario, con el que trabajaremos en adelante, crearemos una base Sqlite. Lo primero que tenemos que hacer es abrir una carpeta nueva para almacenar el fichero que contendrá la base de datos. Para ello, desde línea de comandos, y en nuestra carpeta personal, podemos hacer:

```
mkdir Bases
```

También podemos crear la carpeta con Konqueror o Nautilus, si así lo preferimos.

Ahora rellenaremos los datos de conexión, tal y como aparece en la **Figura 5**.

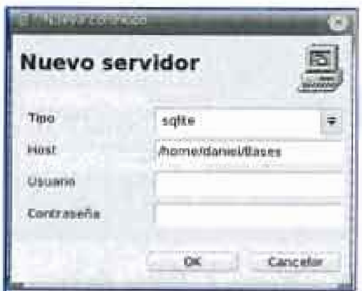


Figura 5. Conexión a una base Sqlite.

Podremos seleccionar *sqlite* o *sqlite3* dependiendo de los gestores instalados en el sistema. Para nuevos desarrollos es recomendable disponer de la versión 3 o posterior, al estar más optimizada. Pero si tenemos que programar para sistemas antiguos (por ejemplo, un cliente que disponga de RedHat 7.0), tal vez debamos plantearnos el usar *sqlite* en sus versiones anteriores, a fin de no tener que compilar e instalar nuevas librerías en el sistema, cuestión que a veces el cliente rechaza o impone.

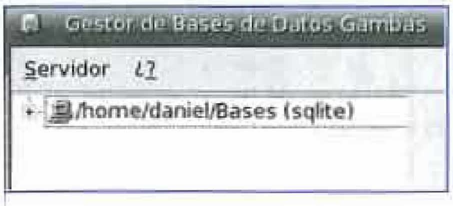


Figura 6. Localización del nuevo servidor.

Una vez incluidos los datos, pulsamos OK y el nuevo servidor quedará reflejado en el árbol de la izquierda del gestor. Si creamos distintos servidores, (distintas carpetas, o bien algunos apuntando a servidores MySQL o Postgres, por ejemplo), se irán añadiendo al árbol.



Figura 7. Menú contextual del servidor

Para conectarnos, haremos doble clic con el botón izquierdo del ratón sobre el servidor (o el botón derecho, si se es zurdo y se tiene así configurado el ratón), y después un clic con el botón derecho para que se muestre el menú contextual de opciones (o izquierdo, para los zurdos).

a opción **Crear usuario...** tiene sentido si trabajamos sobre un sistema servidor de ases de datos (*mysql* o *postgres*), y tenemos permisos de administración sobre el seridor. El uso de esta opción es trivial: se pregunta el nombre de usuario y contraseña y si a su vez tiene permisos de administración; se pulsa **OK** para añadirlo. Más delante veremos algún ejemplo concreto. Seleccionamos la opción **Crear base.** Nos preguntará el **Nombre** de la base, que en el caso de Sqlite será el nombre del archivo dentro de la carpeta que previamente indicamos, el cual alojará la base

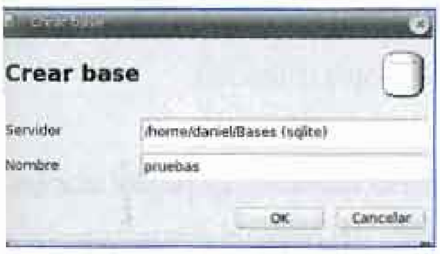


Figura 8. Opción **Crear base.**

de datos que vamos a crear. Señalamos como nombre *pruebas* y pulsamos **OK**. Ya disponemos de una base vacía, donde hemos de crear las distintas tablas que alojarán los datos.

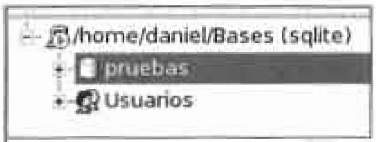


Figura 9. Base **pruebas**.

Crear una tabla

Para crear una tabla, como en el caso del servidor, hacemos doble clic para abrir la base y pulsamos el botón derecho para obtener su menú contextual (**Figura 10**).

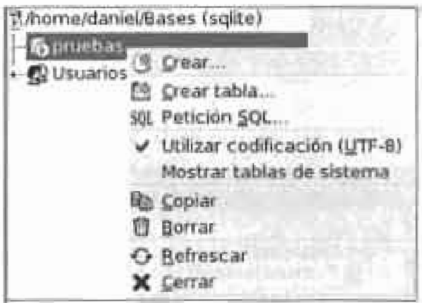


Figura 10. Menú contextual de la base.

Seleccionamos **Crear tabla...** Los datos que se preguntan son el **Nombre** de la tabla (ponemos *datos* en nuestro ejemplo), y el **Tipo**.

Este segundo parámetro sólo tiene sentido en determinados sistemas servidores, como MySQL, en los cuales se puede indicar varios tipos de tablas, optimizadas para una u otra tarea concreta. Salvo que tengamos necesidades especiales, podemos dejar los datos con los valores por defecto.



Figura 11. Ventana para crear una tabla.

Una vez pulsemos sobre el botón **OK**, el gestor aparece en el lado derecho mostrándonos la estructura actual de nuestra tabla *datos* (vacía), para que añadamos la información a los campos que aparecen.

Por tanto, vamos a manejar primero la pestaña **Campos**, con la que crearemos una tabla para almacenar datos de los libros de nuestra propia colección.

Los campos serán: un identificador único, un número que a su vez pegaremos en el lomo (forrado) para buscarlo rápidamente, el título del libro, el autor, la fecha de adquisición, el precio que pagamos por él y una breve descripción.

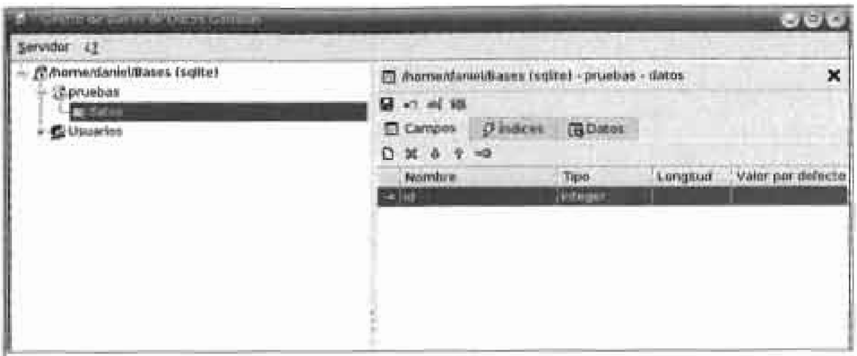


Figura 12. Gestor de la tabla.

Hemos de pensar en los tipos de datos que almacenaremos en cada campo:

- **Identificador:** un número entero.
- **Título:** una cadena de texto.
- **Autor:** una cadena de texto.
- **Fecha:** un dato tipo *Fecha*.
- **Precio:** un número real.
- **Descripción:** una cadena de texto.

> Pensar en los tipos de datos nos evitará, posteriormente, problemas a la hora de hacer búsquedas, ordenar los datos por distintos parámetros, así como a reducir el tamaño de la base de datos y optimizar la velocidad de consulta. Este paso es muy importante y merece la pena dedicarle un tiempo.

También hemos de delimitar la clave principal, que es única y sirve para identificar cada registro almacenado. La clave única estará formada por varios campos o uno sólo. En nuestro caso se trata del campo referido al número de identificador.

Algunos programadores en otros entornos solían diseñar tablas sin claves únicas, dado que algunos sistemas de bases de datos lo permiten. Como resultado se generaban tablas propensas a errores, con datos duplicados, lentas de manejar y muy difíciles de gestionar, conforme pasaba el tiempo y los datos aumentaban, corrompiéndose a veces por duplicaciones. Jamás se debe trabajar de este modo: se rompe toda ventaja que puede aportar una base de datos relacional, y se hipoteca la futura expansión y mantenimiento de una aplicación.

Para crear el primer campo, modificaremos el que el propio gestor ha creado como sugerencia, indicando como **Nombre** *identificador* y **Tipo** *Integer* (es decir, número entero). Dejaremos los demás campos en blanco.

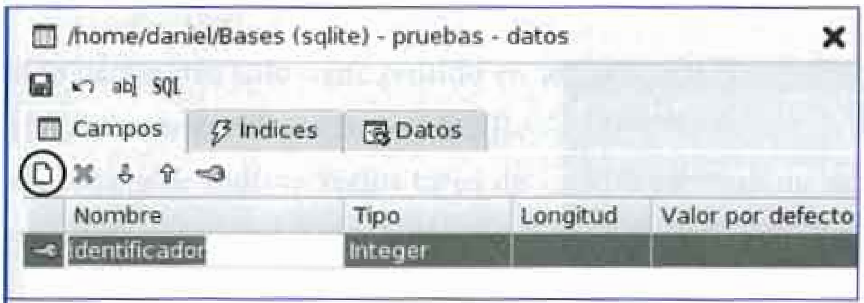


Figura 13. Creación del campo **Identificador**.

Para el resto de campos, pulsamos el botón con el icono que representa una hoja en blanco, y que sirve para añadir un nuevo campo. Si nos movemos por los distintos iconos, aparece un texto de ayuda que nos indica el significado de cada uno.

Conforme los creemos, rellenamos los datos con estos valores:

- **Nombre:** *titulo*; **Tipo:** *String*; **Longitud:** *40*; **Valor por defecto:** (nada).
- **Nombre:** *autor*; **Tipo:** *String*; **Longitud:** *40*; **Valor por defecto:** (nada).
- **Nombre:** *fecha*; **Tipo:** *Fecha*; **Longitud:** (nada); **Valor por defecto:** (nada).

- **Nombre:** *precio*; **Tipo**: *Float*; **Longitud:** (nada); **Valor por defecto:** (nada).

- **Nombre:** *descripcion*; **Tipo:** *String*; **Longitud:** *200*; **Valor por defecto:** (nada).

La interfaz deberá de quedar con este aspecto:

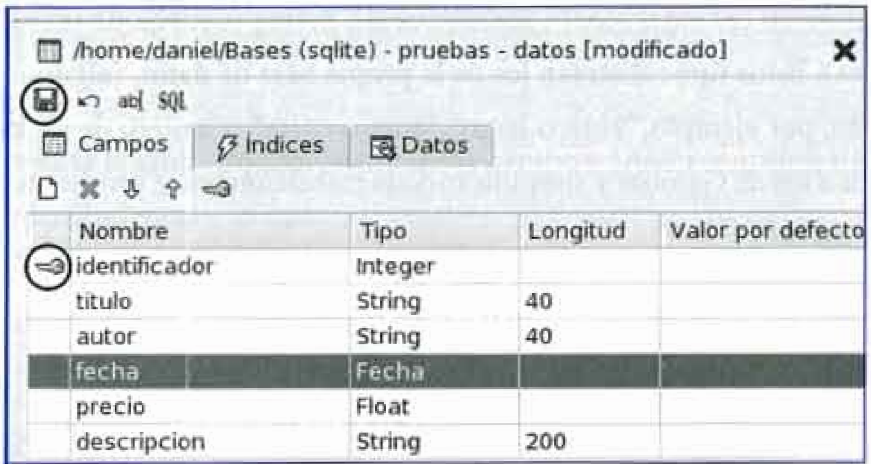


Figura 14. Aspecto final de la creación de los campos.

El dato **Longitud** se aplica sólo a los campos de tipo *String*, y se refiere al número de caracteres que se pueden almacenar. Conviene ajustar lo más posible el valor para no hacer crecer a la base de datos en demasía y, por tanto, hacer más lentas las búsquedas y modificaciones posteriores. Aunque sólo guardemos un carácter en un campo de un registro dado, los sistemas de bases de datos habituales, guardan todo el tamaño indicado (en nuestro caso 40 caracteres para el título y el autor, o 200 para la descripción). Si multiplicamos estos valores por un gran número de registros, comprobaremos rápidamente el ahorro que se consigue con un diseño racional previo de los tamaños.

La longitud máxima de un campo de texto suele ser 200 o 255 caracteres, en muchos sistemas de bases de datos. Por ello, debemos tener en cuenta los tamaños máximos si planeamos cambiar de gestor de bases de datos en un futuro, sobre una aplicación ya desarrollada.

Cuando creamos un nuevo registro, el campo se rellena, por defecto, con un valor, que sólo modificaremos para casos específicos. Esto es útil si un dato se repite muchas veces, y sólo cambia ocasionalmente.

Como hemos podido observar, Gambas realiza una abstracción de los tipos de datos, permitiendo elegir entre unos muy concretos: *Boolean* (dos valores, "sí" o "no"), *Integer* (número entero), *Float* (número real), *String* (cadena de texto) y *Fecha* (fecha/hora). Estos tipos abstraen los de la propia base de datos, (algunos sistemas contemplan, por ejemplo, "entero largo", "entero corto" o "entero de un byte") asimilándolos a los de Gambas y simplificando la transición entre un sistema de bases de datos y otros.

Si existe necesidad de optimizar los tipos de datos para un sistema concreto, las tablas se deben crear con las herramientas propias de ese sistema de base de datos, en lugar de la herramienta genérica de Gambas, pero a cambio se pierde portabilidad. Es decir, será más difícil cambiar de sistema gestor en el futuro, si es necesario por el aumento de volumen de datos, tecnología o velocidad requeridas. En general, no es necesario recurrir a optimizaciones de este nivel, salvo en casos muy concretos.

En cuanto a la clave principal, podemos observar en la **Figura 14** que el primer campo, *identificador*, tiene una llave amarilla marcada. Podemos pulsar sobre el primer elemento de cada campo para que aparezca o desaparezca dicha llave.

La suma de todos los campos que estén marcados con la llave, conforma la clave principal o identificador único de cada registro. Aunque en nuestro caso, sólo tenemos uno.

Estos datos, hasta ahora, se encuentran sólo en fase de diseño. Para grabar la nueva tabla en la base de datos, pulsaremos el icono que tiene el símbolo de un **floppy** (disquete). A partir de este momento, la tabla aparece realmente en la base, lista para añadir datos o consultarlos.

Los nombres de campos usados en el ejemplo no tienen acentos. En general, no se debe utilizar otra cosa que caracteres del alfabeto inglés y números. Usar símbolos como caracteres con tilde, la eñe o la cedilla, separadores tales como el ampersand (&) o un espacio en blanco, pueden dar muchos dolores de cabeza al programador en el futuro, tanto para una simple consulta de datos, como para la migración futura a otro sistema de bases de datos. Es una mala política de diseño, hay que evitarlos siempre.

Una vez creada la tabla, podemos editar los campos, añadir o quitar, siguiendo el mismo procedimiento.

Ahora podemos pasar a la siguiente pestaña (**Figura 15**), **Índices**. Un índice sirve al gestor de bases de datos para organizar la información, de forma que más tarde cada consulta se ejecute lo más rápido posible. Por ejemplo, puede que deseemos hacer frecuentemente búsquedas indicando el autor, para saber los datos de todos sus libros que tenemos en nuestra colección.

Para nuestra pequeña base de ejemplo no utilizaremos índices, pero su gestión desde este programa es trivial: basta con que los creemos, eliminemos o modifiquemos del mismo modo que hemos hecho para crear los campos de las tablas. Cada índice puede estar formado por uno o más campos, es decir, puede comprender, por ejemplo, el nombre del libro + el autor. A tal efecto, la interfaz proporciona dos botones. El primero, **Nuevo índice**, sirve para añadir un índice a la tabla. Si está formado sólo por un campo, basta con que indiquemos el nombre de ese campo. Si hay varios, se pulsará el botón **nuevo campo de índice**, tantas veces como sea necesario, para indicar, a continuación, los demás campos que conforman ese índice.

El otro dato de relevancia es el valor **Unique**, el cual determina si pueden existir o no dos registros en los cuales los datos de los campos pertenecientes a un índice sean iguales. Por ejemplo, en una tabla que contiene datos de CDs, puede interesarnos tener un índice creado a partir de los campos *discografica* y *fecha*. Y puede haber varios discos editados por esa discográfica en un mismo año, por tanto marcaremos

el valor **Unique** a *FALSE*, para poder introducir los datos de varios discos cuyos campos sean coincidentes.

Por el contrario, en una base de datos de alumnos puede interesarnos un índice que comprenda el DNI y su nombre. En este caso, no hay dos personas con el mismo DNI, luego ese índice es único, y marcaremos la casilla **Unique** con el valor *TRUE* para proteger a la tabla de datos duplicados.

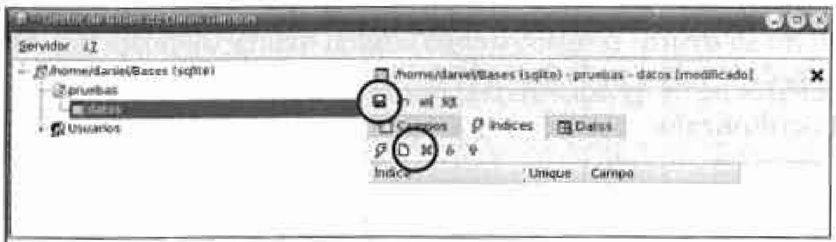


Figura 15. Creación de **Índices**.

Gestionar datos de una tabla

La última pestaña de la interfaz del gestor de bases de datos, **Datos**, nos permite consultar y modificar el contenido de la tabla. Dispone de tres botones: **Nuevo registro** (hoja en blanco), **Borrar registro** (aspa roja) y **Guardar datos** (floppy). Hay que tener siempre presente que, al trabajar con esta interfaz, los datos que añadamos, eliminemos o modifiquemos, sólo se actualizan realmente en la base de datos cuando pulsamos el botón para guardar.

Cada vez que pulsamos el botón **Nuevo registro**, se crea una línea, correspondiente a un nuevo registro en la tabla actual. Podemos desplazarnos por los distintos campos de ese registro, para añadir los datos correspondientes. También, si la tabla contiene ya datos, podemos modificarlos o bien podemos seleccionar un registro y eliminarlo con la opción **Borrar registro**.

En este último caso el registro se pone de un color distinto para señalar que en la próxima actualización dicho registro desaparecerá, lo cual nos permite asegurarnos que la solicitud de eliminación es correcta antes de pulsar el botón **Guardar datos**.

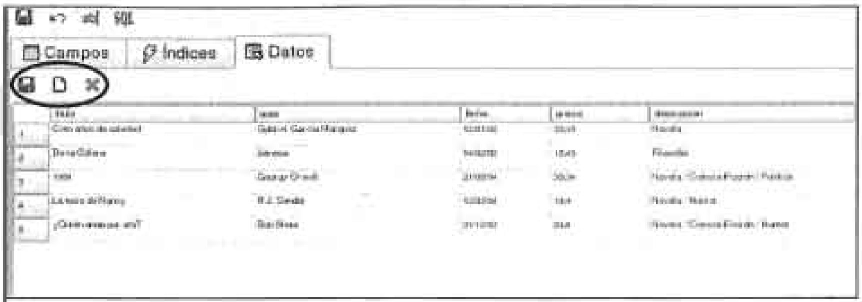


Figura 16. Pestaña de **Datos**.

Introduciremos registros de libros utilizando esta interfaz, para que dispongamos de algunos datos con los que trabajar en adelante. Recordemos que es importante pulsar el botón **Guardar datos** cuando hayamos terminado la edición.

□□□□□ SQL

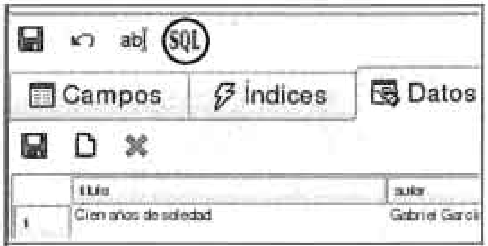


Figura 17. Opción **SQL**.

Sin escribir aún código Gambas, podemos ya empezar a trabajar con el lenguaje SQL. Como hemos podido observar, uno de los botones del gestor de bases de datos de Gambas contiene un texto SQL. Lo pulsaremos para escribir sentencias en este lenguaje.

Nuestra primera consulta servirá para obtener un listado de todos los datos de la tabla que hemos creado. Para los que no estén muy familiarizados con el lenguaje SQL, aclaramos los siguientes puntos: las instrucciones de consulta comienzan con la instrucción *SELECT*, a continuación hay que determinar qué queremos consultar (en nuestro caso todos los campos, para lo cual emplearemos el símbolo *) y después se usa la palabra clave *FROM* para indicar de dónde obtener esos datos.

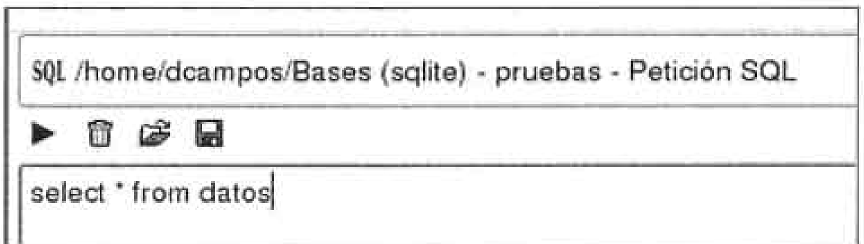
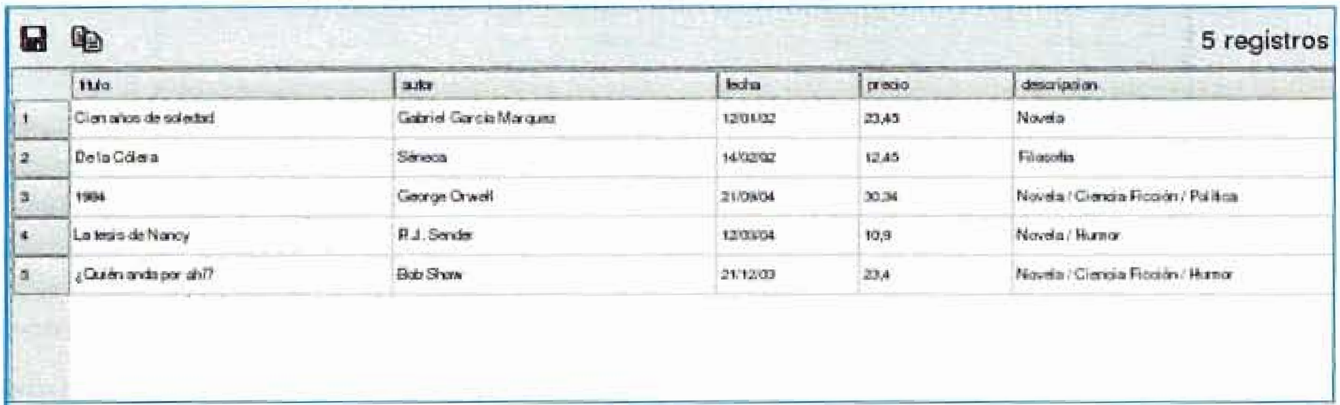


Figura 18. Lenguaje SQL.

Al pulsar el botón **Ejecutar**, obtenemos el resultado de la consulta en la parte inferior.



5 registros

	titulo	autor	fecha	precio	descripcion
1	Cien años de soledad	Gabriel García Marquez	12/01/02	23,45	Novela
2	De la Cólera	Séneca	14/02/02	12,45	Filosofía
3	1984	George Orwell	21/09/04	30,34	Novela / Ciencia Ficción / Política
4	La tesis de Nancy	R.J. Sender	12/03/04	10,9	Novela / Humor
5	¿Quién anda por ahí?	Bob Shaw	21/12/03	23,4	Novela / Ciencia Ficción / Humor

Figura 19. Resultado de la consulta.

Podemos guardar las consultas complejas en un archivo y luego recuperarlas para su uso posterior, con los botones **Guardar petición** y **Cargar petición**. También podemos exportar los resultados de la consulta con el botón **Exportar datos**, o copiarlo al portapapeles para pegarlo en una hoja de cálculo como Gnumeric, Kcalc u OpenOffice Calc.

Para familiarizarnos con SQL, también podemos realizar las siguientes consultas:

1. Mostrar el título y autor, ordenado por el nombre de autor:

```
select autor,titulo from datos order by autor
```

2. Obtener todos los títulos de los libros cuyo precio es mayor de 15 euros:

```
select titulo from datos where precio>15
```

3. Obtener todos los datos de los libros adquiridos antes del 2004:

```
select * from datos where fecha<"2004-01-01"
```

Los formatos para representar fechas y caracteres genéricos (comodines), varían de una base de datos a otra, y puede resultar que una consulta dada sólo funcione para un sistema de bases de datos determinado.

4. Obtener todos los títulos que contienen la palabra *la*:

```
select * from datos where titulo like "%la%"
```

Con esta interfaz no sólo podemos consultar, sino que también podemos realizar cualquier operación permitida por SQL+Sistema Gestor de Bases de Datos. Por ejemplo, podemos añadir un nuevo registro en la tabla **datos**:

```
insert into datos values ("El lazarillo de Tormes",
"Anónimo",2005-02-03,20.91,"Novela")
```


Al utilizar instrucciones *insert, delete* o cualquier otra que no dé como resultado un grupo de registros, el gestor muestra la leyenda *No hay registros,* si tuvimos éxito, o bien un mensaje de error si la base de datos no aceptó el comando.

5. 4 Programación

Modelo de bases de datos

Gambas, a través del componente *gb.db*, realiza una abstracción de la estructura y los datos de una base. El primer requisito necesario para esta labor es saber cómo acceder o conectarse a una base de datos.

De esta tarea se encarga la clase *Connection*. Creando varios objetos *Connection* podemos conectarnos a varias bases de datos.

El objeto *Connection* dispone de una serie de propiedades para determinar el tipo de sistema de bases de datos, el nombre de la base, la ubicación del recurso (puede ser una ruta en el sistema de archivos o una dirección IP), y los datos del usuario que desea conectarse a dicho sistema. Esta clase, junto con la clase *Database*, aportan la información necesaria sobre nuestra base de datos. La clase *User* perfila nuestro usuario y, por tanto, nuestros permisos de acceso.

La información de una base de datos se almacena en diversas tablas. Cada tabla se puede imaginar como una rejilla bidimensional, al estilo de una hoja de cálculo. En horizontal tenemos los encabezados que nos indican la información de cada campo de nuestra tabla. En Gambas, el componente *gb.db* aporta una clase *Table*, que define la estructura de una tabla, y la clase *Field* que determina las características de un campo de una tabla.

En vertical se van acumulando los distintos registros con información. Cuando se trabaja con bases de datos relacionales, el programador no se limita a hojear el contenido de una tabla, sino que, empleando instrucciones SQL, puede consultar promedios de valores, sumas, uniones coherentes de los datos de varias tablas, datos agrupados por una clave u ordenados según algún criterio, etc.

En todo caso, el sistema de bases de datos devuelve siempre el resultado de la consulta como una serie de registros que disponen de varios campos, es decir, en un formato similar al de las propias tablas. Los objetos de la clase *Result* proporcionan el acceso a dichos registros y campos de información, como si se tratara de una tabla.

La clase *ResultField* es similar a *Field*, pero en este caso no proporciona información sobre un campo de una tabla de una base de datos, sino sobre un campo de un conjunto de registros procedentes de una consulta.

□□□□□ Conectándose por código

Vamos a conectarnos a la base de datos Sqlite que creamos en el primer apartado. Dependiendo de nuestra versión de Sqlite, usaremos el driver llamado *sqlite* o el *sqlite3*.



Figura 20. Formulario **FMain**.

Para ello creamos un proyecto gráfico llamado *MisLibros*. Situamos en él un formulario llamado *FMain*, que será el de arranque, y, dentro de él, un control ColumnView llamado *Tabla*. Este control ColumnView mostrará los datos de la tabla que habíamos creado en los ejercicios anteriores.

Ahora vamos a escribir una función que se conecte a la base de datos dentro del código del formulario:

```
Private hConn as Connection

PRIVATE FUNCTION ConectarBase() AS Boolean

  IF hConn<>NULL THEN RETURN FALSE

  hConn = NEW Connection

  hConn.Host = "/home/usuario/Bases"
  hConn.Name = "pruebas"
  hConn.Type = "sqlite"
  TRY hConn.Open()
  IF ERROR THEN
    hConn = NULL
    Message.Error("Error al conectar con la base")
    RETURN TRUE
  END IF

  RETURN FALSE

END
```

Definimos un objeto **Connection** que será accesible en todo el formulario y representa la conexión a nuestra base de datos. Después se escribe una función **ConectarBase** que devuelve **FALSE** si tiene éxito o **TRUE** si falló.

El código conlleva estos pasos:

1. Si ya hay una conexión (si el objeto **hConn** no es nulo), retornamos inmediatamente indicando que ya hay conexión **FALSE**, de esta forma podremos llamar sistemáticamente a esta función desde varios puntos sin preocuparnos de si ya existe la conexión o no.

2. Creamos el objeto **Connection**, que en principio no está conectado a ninguna base, y proporcionamos la información necesaria para conectar: rellenamos la propiedad **Host**, que en el caso de una base de datos con servidor podría ser una dirección IP, pero que para Sqlite es una ruta dentro del sistema de archivos (recuerda sustituir */home/usuario* por la ruta donde tú almacenaste la base del ejemplo); después indicamos el nombre de la base de datos (**pruebas**), que en el caso de Sqlite es también el nombre del archivo que está dentro de la carpeta **/home/usuario/Bases**; y por último indicamos el tipo de base de datos al que nos conectaremos, y que será **sqlite** o **sqlite3**, de acuerdo con el diseño que realizamos con el Gestor de Bases de Datos de Gambas. Si nuestra conexión se realizará sobre un servidor MySQL, por ejemplo, también tendríamos que rellenar las propiedades *Login* y *Password*, con el nombre de usuario y contraseña de acceso al servidor.

Puesto que aún no estamos realmente conectados, podemos rellenar todos estos datos en el orden que deseemos.

3. Después, tratamos de abrir la conexión. Si no fuera posible (por ejemplo, por una ruta incorrecta o un fallo en el servidor), se disparará un error, que capturamos con la orden **TRY**.

4. Si se ha producido un error, hacemos nula de nuevo la conexión fallida y retornamos con el valor de error, que según hemos decidido, es **TRUE**.

5. Si, por el contrario, hubo éxito, retornamos el valor correspondiente (**FALSE**).

A continuación, vamos a crear una función a la que llamar para cerrar la conexión. El algoritmo es sencillo, si no hay ninguna conexión, retornamos, en caso contrario, cerramos la conexión y la hacemos nula para que una próxima llamada a *ConectarBase* se aperciba de que no hay una conexión activa.

```
PRIVATE SUB CerrarConexion()

  IF hConn = NULL THEN RETURN
  hConn.Close()
  hConn = NULL

END
```


Consulta de datos

Al abrirse el formulario, abriremos la conexión, leeremos los datos disponibles, los representaremos en nuestro control *Tabla* y cerraremos la conexión. Esto quedaría representado de la siguiente manera:

```
PUBLIC SUB Form_Open()

  DIM hResul AS Result
  DIM Clave AS String

  Tabla.Clear()

  IF ConectarBase() THEN RETURN

  Tabla.Columns.Count = 5
```

```
    Tabla.Columns[0].Text = "Título"
    Tabla.Columns[1].Text = "Autor"
    Tabla.Columns[2].Text = "Fecha"
    Tabla.Columns[3].Text = "Precio"
    Tabla.Columns[4].Text = "Descripción"

    hResul = hConn.Exec("select * from datos")

    DO WHILE hResul.Available

      Clave = hResul["titulo"]

      Tabla.Add(Clave, Clave)

      Tabla[Clave][1] = hResul["autor"]
      Tabla[Clave][2] = hResul["fecha"]
      Tabla[Clave][3] = hResul["precio"]
      Tabla[Clave][4] = hResul["descripcion"]

      hResul.MoveNext()

    LOOP

    CerrarConexion()

END
```


El código trata de abrir la conexión y si fracasa sale de la función. Si todo va bien, define cinco columnas en nuestro control de tabla, pone el título a los encabezados de columna y procede a rellenar los diferentes registros. Para efectuar esta tarea, primero consultamos los datos de la tabla, empleando el método **Exec** de nuestro objeto **hConn**, al cual pasamos como parámetro la consulta SQL y nos devuelve un objeto de la clase *Result*, que contiene cada uno de los registros provenientes de la consulta.

El objeto *Result* tiene un puntero interno que en cada momento apunta a uno de los registros. En el estado inicial apunta al primer registro, si es que hay alguno. Este objeto dispone de una serie de métodos para mover el puntero. **MoveNext** (el de nuestro ejemplo) lo mueve al siguiente registro, si existe; *MovePrevious*, al anterior; *MoveFirst*, al primero; y *MoveLast*, al último. Con *MoveTo* podemos especificar un registro concreto al que desplazarnos.

Cuando estamos situados en un registro, **Result** nos permite obtener el dato correspondiente a un campo, indicando el nombre del campo como si el objeto fuese un Array: en nuestro ejemplo, **hResul["autor"]** nos devuelve el valor del campo *autor* en el registro actual.

Si como resultado de un movimiento del puntero hemos sobrepasado el último registro o estamos antes del primero, la propiedad **Available** toma el valor *False*, mientras que si nos encontramos apuntando a un registro, **Available** toma el valor *True*. Igualmente, si no hay registros resultantes de la consulta, **Available** valdrá *False*. (Available significa *Disponible* en inglés).

Con estos conocimientos ya podemos realizar un bucle para rellenar los datos de nuestra tabla.

Mientras la propiedad **Available** sea *True*, tomamos como clave el valor del campo **"titulo"**, añadimos una línea en nuestra tabla identificada por esa clave, y cuyo texto (la primera columna) es dicha clave. Rellenamos el resto de los campos con los valores provenientes del objeto **hResul**, y nos movemos al siguiente registro. Tras salir del bucle, cerramos la conexión y abandonamos la función.

Como último retoque antes de ejecutar, ponemos la propiedad Sorted del control ColumnView a *True*, de modo que el usuario, al pulsar sobre un encabezado de columna, pueda ordenar el listado por el campo que prefiera. Puesto que cada registro en el listado está identificado por la misma clave que tiene en la tabla, posteriores operaciones de selección/modificación por parte del usuario, no se ven afectadas por el orden de representación.

El resultado final es éste:

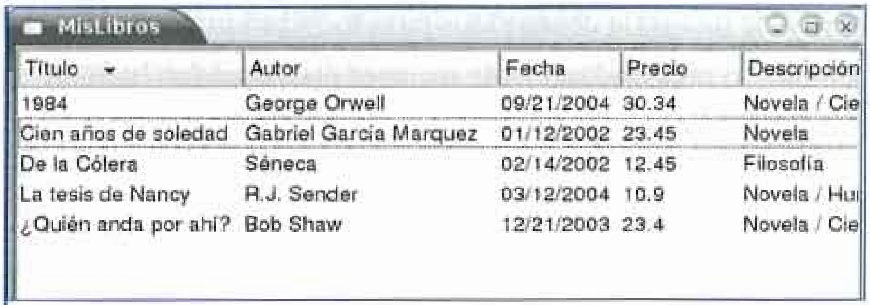


Figura 21. Aspecto final del proyecto MisLibros.

Aprovechar las posibilidades de ordenación de los controles, puede evitar sobrecargar al servidor o al cliente de bases de datos con múltiples consultas.

Borrar registros

Podemos añadir algo de código para borrar registros. Es decir que cuando el usuario pulse la tecla **Supr** o **Del**, el registro que esté seleccionado se borre.

```
PUBLIC SUB Tabla_KeyRelease()

  IF Key.Code = Key.Delete THEN

    IF Tabla.Current = NULL THEN RETURN
    IF Tabla.Current.Selected = FALSE THEN RETURN

    IF ConectarBase() THEN RETURN

    TRY hConn.Exec("delete from datos where titulo=&1",
    Tabla.Current.Key)
    IF ERROR THEN
      Message.Error("Imposible borrar el registro")
    ELSE
```

```
        Table.Current.Delete()
        END IF

        CerrarConexion()

      END IF

    END
```

Aprovechamos el evento **Key_Release** de nuestro control **Tabla.** Si la tecla pulsada es **Del** o **Supr**, primero comprobamos si existe algún elemento actual en la tabla; si no es así, salimos. Si el elemento actual no está seleccionado, también salimos. Por otro lado, tratamos de abrir la conexión, y si hay éxito, ejecutamos una instrucción SQL para borrar un registro cuya clave coincida con la del registro de nuestro control *ColumnView*. Si se produce un error (por ejemplo, una base de sólo lectura), se informa al usuario. En caso contrario, se borra también el registro del control para reflejar el cambio. Cerramos la conexión y salimos de la función.

En general, es preferible usar *KeyRelease* y *MouseRelease*, a los equivalentes *Press*, cuando se ejecuta una acción por parte del usuario. De este modo, cuando ha pulsado el botón del ratón, aún tiene tiempo para apartarlo del control y cancelar así una acción errónea, antes de levantar el dedo del ratón. De otro modo, en el evento *MousePress* podría haber borrado un registro sin tiempo para reaccionar.

Como podemos observar, en el caso siguiente la sentencia SQL está construida de un modo algo diferente. Tenemos dos parámetros, el primero es parte de la consulta y el segundo contiene el elemento concreto al que nos referimos:

```
hConn.Exec("delete from datos where titulo=&1",
Tabla.Current.Key)
```

El método **Exec** admite un número variable de parámetros, y a la hora de tratar la sentencia SQL, Gambas sustituye los indicadores de parámetros (&1, &2, &3...) por el parámetro que corresponde en orden (&1 es el primer parámetro extra, &2 el segundo...). Además, formatea el dato según se necesite (por ejemplo, este parámetro corresponde a un dato de tipo *String*, que en la sintaxis Sqlite requiere unas comillas antes y detrás). Esto es muy útil para trabajar con datos de tipo fecha/hora, cadenas de texto y números con decimales, ya que la sintaxis puede variar de un tipo de bases de datos a otras, e incluso con la misma base puede cambiar según los parámetros regionales (formato de fecha, usar coma o punto para decimales, etc.). Con este sistema, pondremos simplemente un dato tipo Cadena, Booleano, Fecha o Número, y Gambas se encargará de dar el formato adecuado.

Dejar a Gambas la tarea de dar formato a los diferentes tipos de datos, evitará muchos problemas si cambiamos de sistema gestor de bases de datos, o pretendemos que la aplicación funcione correctamente en varios PC que puedan tener diferentes configuraciones.

□□□□□ Añadir registros

Vamos a añadir ahora un formulario llamado **FData** que nos servirá para introducir datos y también, en el siguiente apartado, para editar los ya existentes. Este formulario tendrá cinco etiquetas, con cualquier nombre. Cinco cajas de texto llamadas **TxtTitulo**, **TxtAutor**, **TxtFecha**, **TxtPrecio** y **TxtDescripcion**, y dos botones llamados **BtnAceptar** y **BtnCancelar**. El aspecto general puede ser como el que vemos en la figura de la derecha (recordemos poner los textos en las etiquetas de acuerdo con la caja de texto que corresponda).

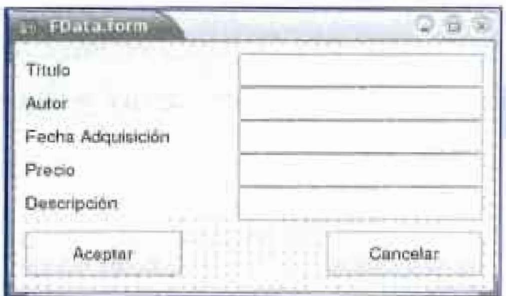


Figura 22. Formulario **FData**.

Este formulario recibirá una referencia a la conexión con la base de datos, por tanto, dispondrá de una referencia a un objeto **Connection**.

Por ello deberemos escribir al principio del código de este formulario lo siguiente:

```
PRIVATE hConn AS Connection
```

Así mismo, dispondrá del método **RunNew** con el que le llamaremos desde el formulario principal, pasando como parámetro nuestra conexión a la base, y que mostrará el formulario de forma modal, a la espera de la introducción de datos por parte del usuario.

```
PUBLIC SUB RunNew(Data AS Connection)

  hConn = Data
  ME.ShowModal()

END
```

El código toma en **hConn** la referencia al objeto **Connection** que le pasemos, y lo muestra. En cuanto al botón de cancelación, es bien sencillo: basta con indicar al formulario que se cierre que de la siguiente manera:

```
PUBLIC SUB BtnCancelar_Click()

  ME.Close()

END
```

En cuanto al botón **Aceptar**, en él realizaremos la operación de alta de un nuevo registro, utilizando la instrucción SQL **insert into**:

```
PUBLIC SUB BtnAceptar_Click()

  TRY hConn.Exec("insert into datos values
  (&1,&2,&3,&4,&5)", TxtTitulo.Text, TxtAutor.Text,
```

```
CDate(TxtFecha.Text), CFloat(TxtPrecio.Text),
TxtDescripcion.Text)

END IF

ME.Close()

CATCH
  Message.Error("Imposible introducir los datos
  solicitados")

END
```

Indicamos al objeto *Connection* que ejecute la sentencia SQL, la cual, para las altas sigue siempre el mismo patrón:

insert into [nombre de la tabla] **values** ([valor del campo 1], [valor del campo 2]....)

Para evitar problemas de formateo, seguimos empleando los parámetros adicionales, como en el caso antes comentado.

Los datos de **Título**, **Descripción** y **Autor** se pasan directamente, al ser datos tipo texto.

El dato del **Precio** se convierte a número real con **CFloat()**, y la **Fecha** se convierte a dato fecha/hora con **CDate()**.

Si se produce un error, bien sea por falta de permisos de escritura o por datos no convertibles a fechas o números, por ejemplo, se captura con TRY, y se trata en el bloque de código CATCH donde indicamos el mensaje de error al usuario.

Si, por el contrario, todo fue bien, se cierra el formulario tras el alta.

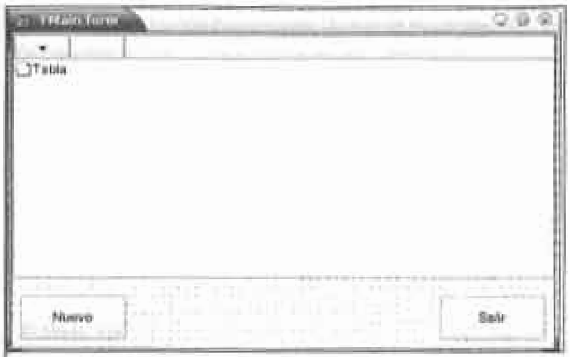


Figura 23. Creación de los botones **Nuevo** y **Salir** del formulario principal.

Volvamos ahora al formulario principal (FMain), y añadamos dos botones para completar la interfaz: uno llamado **BtnAlta**, para dar de alta un nuevo registro, y otro **BtnCerrar**, para cerrar el programa. Como siempre, el botón de salida es sencillo de implementar (recordemos que ahora escribimos el código en *Fmain*, no en *Fdata*):

```
PUBLIC SUB BtnSalir_Click()

  ME.Close()

END
```

Para el botón de nuevo registro, como siempre, tratamos de conectar a la base, pero nos salimos si no lo logramos. A continuación llamamos al método **RunNew** de **FData**. Puesto que se muestra de forma modal, el código de este formulario queda aquí detenido hasta que el usuario dé un alta o cierre el formulario.

Tras esto, y de nuevo en nuestra función, cerramos la conexión y llamamos al método **Form_Open()**, que, como recordaremos, se encargaba de representar los datos en la tabla, para reflejar así el alta dada por el usuario.

```
PUBLIC SUB BtnNuevo_Click()

  IF ConectarBase() THEN RETURN
  FData.RunNew(hConn)
  CerrarConexion()
  Form_Open()

END
```

Ya disponemos de una interfaz básica para dar altas. Se pueden mejorar muchas cosas. Entre otras, se pueden bloquear los cuadros de texto para que sólo acepten números y el signo decimal (en el caso del precio), para que sólo acepte un dato válido como fecha (en el caso del campo **Fecha**), y que compruebe la longitud máxima en el caso de los diferentes campos de texto. También se puede ajustar el formato de fecha al adecuado para el usuario (aquí usamos directamente el de Sqlite, es decir, dos dígitos de mes / dos dígitos de día / dos dígitos de año), o traspasar los datos de vuelta al formulario principal para que sólo se actualice el nuevo registro en lugar de toda la tabla completa.

□□□□□ Modificar registros

Aprovecharemos el formulario de altas para las modificaciones. Cuando el usuario haga doble clic sobre uno de los registros, aparecerá el formulario de altas, con los datos del registro actual, pero esta vez en modo modificación para que el usuario cambie sólo lo que quiera. Modificaremos el código del formulario de datos para que disponga de un flag para indicar si trabaja sobre un alta o modificación, y dispondremos también de una referencia a un objeto de la clase **Result**, que apuntará al registro actual a modificar. El principio del código del formulario Fdata quedaría así:


```
PRIVATE Editando AS Boolean
PRIVATE hResul AS Result
PRIVATE hConn AS Connection
```

En el mismo formulario añadiremos un nuevo método que, a diferencia de *RunNew* prepara al formulario para entrar en modo edición. Para ello recibimos la referenci: al objeto de la clase **Result**; ponemos el flag **Editando** a **TRUE**; y ponemos en cad: caja de texto el valor del campo correspondiente. Tras esto, se muestra de forma modal

```
PUBLIC SUB RunEdit(Data AS Result)

  hResul = Data
  Editando = TRUE
  TxtTitulo.Text = hResul["titulo"]
```

```
    TxtAutor.Text = hResul["autor"]
    TxtFecha.Text = hResul["fecha"]
    TxtPrecio.Text = hResul["precio"]
    TxtDescripcion.Text = hResul["descripcion"]
    ME.ShowModal()

END
```

Hemos de modificar también el código del botón **BtnAceptar** para que distinga entre altas y modificaciones, y actúe en consecuencia. En el caso de la modificación, sitúa el valor de cada caja de texto en el campo correspondiente, y de existir un error, se tratará en la zona CATCH al estar protegida cada inserción con una instrucción TRY. Si hubo éxito, se llama al método **Update**, que hace que los valores que hemos introducido en los campos se actualicen realmente en la base de datos. Tras esto, como en el caso del alta, se cierra y descarga el formulario.

```
PUBLIC SUB BtnAceptar_Click()

  IF Editando THEN
    TRY hResul["titulo"] = TxtTitulo.Text
    TRY hResul["autor"] = TxtAutor.Text
    TRY hResul["fecha"] = TxtFecha.Text
    TRY hResul["precio"] = TxtPrecio.Text
    TRY hResul["descripcion"] = TxtDescripcion.Text
    TRY hResul.Update()
  ELSE

    TRY hConn.Exec("insert into datos values
    (&1,&2,&3,&4,&5)", TxtTitulo.Text, TxtAutor.Text,
    CDate(TxtFecha.Text), CFloat(TxtPrecio.Text),
    TxtDescripcion.Text)

  END IF
```

```
    ME.Close()

  CATCH
    Message.Error("Imposible introducir los datos
    solicitados")

  END
```

Añadir nuevas funcionalidades a un elemento ya existente, como un formulario, puede ahorrar tiempo y código, pero en ocasiones puede hacer el nuevo código complejo y difícil de mantener. Hay que sopesar el equilibrio entre coste de programación y mantenimiento posterior en cada caso, antes de crear una nueva estructura de código o modificar la existente.

De nuevo en el formulario principal, *FMain*, aprovecharemos el evento **Activate** del control *ColumnView*, que se genera al hacer doble clic, para dar acceso a esta nueva funcionalidad.

```
PUBLIC SUB Tabla_Activate()

  DIM hResul AS Result

  IF Tabla.Current = NULL THEN RETURN

  IF ConectarBase() THEN RETURN

  hResul = hConn.Edit("datos", "titulo=&1",
  Tabla.Current.Key)
  FData.RunEdit(hResul)

  Tabla.Current[0] = hResul["titulo"]
```

```
    Tabla.Current[1] = hResul["autor"]
    Tabla.Current[2] = hResul["fecha"]
    Tabla.Current[3] = hResul["precio"]
    Tabla.Current[4] = hResul["descripcion"]

    CerrarConexion()

END
```

Como podemos observar, ahora el objeto **Result** no lo obtenemos como llamada al método *Exec* si no con otro nuevo método: **Edit**. La razón es que las llamadas a *Exec*, que reciben como parámetro una sentencia SQL, son de sólo lectura, es decir, es posible examinar el contenido de un campo como por ejemplo:

```
Cadena=hResul["titulo"]
```

Pero no es posible actualizar el contenido del campo, es decir, no se puede realizar:

```
hResul["titulo"]=Cadena
```

que es precisamente lo que nos interesa.

El método **Edit** no recibe como parámetro una sentencia SQL, sino el nombre de la tabla que se desea modificar, y como segundo parámetro y posteriores, un filtro y los parámetros de ese filtro. Es fácil entender el modo en que trabaja si partimos de una instrucción SQL y obtenemos las partes que nos interesan para el método **Edit**.

Supongamos que de una tabla **Articulos** queremos modificar todos aquellos cuyo campo **precio** sea superior a **20**. Una sentencia SQL para obtener ese conjunto de registros sería:

```
select * from Articulos where precio>20
```

Para construir la llamada al método **Edit** tendríamos:

Nombre de la tabla: Articulos.
Filtro: precio mayor que una determinada cantidad.
Parámetros del filtro: 20, en este caso.

Por tanto la llamada queda como:

```
hConn.Edit (  "Articulos" , "precio>&1", 20 )
```

¿Por qué no se emplean sentencias SQL directamente? Porque las modificaciones tienen sentido sólo sobre una tabla. No se puede modificar, por ejemplo, el resultado de los registros que provienen de unir dos tablas, o de un sumatorio de los valores de un campo.

Tras obtener el registro con la llamada a **Edit**, pasamos el objeto *Result editable* al formulario de datos, con la llamada a su método *RunEdit*, y éste se mostrará de forma modal. Al cerrarse el formulario, la ejecución retorna a nuestro método en el cual actualizamos el contenido de nuestro control *ColumnView* para reflejar los cambios hechos en la tabla de la base.

A continuación, se reproducen completos el código de *FMain* y *FData* al ser dos códigos extensos que hemos troceado y modificado sobre la marcha.

FMAIN

```
PRIVATE hConn AS Connection

PRIVATE FUNCTION ConectarBase() AS Boolean

  IF hConn THEN RETURN FALSE

  hConn = NEW Connection
```

```
  hConn.Host = "/home/usuario/Bases"
  hConn.Name = "pruebas"
  hConn.Type = "sqlite"
  TRY hConn.Open()
  IF ERROR THEN
    hConn = NULL
    Message.Error("Error al conectar con la base")
    RETURN TRUE
  END IF

  RETURN FALSE

END


PRIVATE SUB CerrarConexion()

  IF hConn = NULL THEN RETURN
  hConn.Close()
  hConn = NULL

END



PUBLIC SUB Form_Open()

  DIM hResul AS Result
  DIM Clave AS String

  Tabla.Clear()

  IF ConectarBase() THEN RETURN
```

```
  Tabla.Columns.Count = 5
  Tabla.Columns[0].Text = "Título"
  Tabla.Columns[1].Text = "Autor"
  Tabla.Columns[2].Text = "Fecha"
  Tabla.Columns[3].Text = "Precio"
  Tabla.Columns[4].Text = "Descripción"
  hResul = hConn.Exec("select * from datos")

  DO WHILE hResul.Available

    Clave = hResul["titulo"]
    Tabla.Add(Clave, Clave)
    Tabla[Clave][1] = hResul["autor"]
    Tabla[Clave][2] = hResul["fecha"]
    Tabla[Clave][3] = hResul["precio"]
    Tabla[Clave][4] = hResul["descripcion"]
    hResul.MoveNext()

  LOOP

  CerrarConexion()

END


PUBLIC SUB Tabla_KeyRelease()

  IF Key.Code = Key.Delete THEN

    IF Tabla.Current = NULL THEN RETURN
    IF Tabla.Current.Selected = FALSE THEN RETURN

    IF ConectarBase() THEN RETURN
```

```
    TRY hConn.Exec("delete from datos where titulo=&1",
    Tabla.Current.Key)
    IF ERROR THEN Message.Error("Imposible eliminar el
    registro")
    CerrarConexion()
    Tabla.Current.Delete()

  END IF

END

PUBLIC SUB Tabla_Activate()

  DIM hResul AS Result

  IF Tabla.Current = NULL THEN RETURN
  IF ConectarBase() THEN RETURN
  hResul = hConn.Edit("datos", "titulo=&1",
  Tabla.Current.Key)
  FData.RunEdit(hResul)
  Tabla.Current[0] = hResul["titulo"]
  Tabla.Current[1] = hResul["autor"]
  Tabla.Current[2] = hResul["fecha"]
  Tabla.Current[3] = hResul["precio"]
  Tabla.Current[4] = hResul["descripcion"]
  CerrarConexion()

END


PUBLIC SUB BtnSalir_Click()

  ME.150%Close()
```

```
END

PUBLIC SUB BtnNuevo_Click()

  IF ConectarBase() THEN RETURN
  FData.RunNew(hConn)
  CerrarConexion()
  Form_Open()

END
```

FDATA

```
PRIVATE Editando AS Boolean
PRIVATE hResul AS Result
PRIVATE hConn AS Connection

PUBLIC SUB RunNew(Data AS Connection)

  hConn = Data

  ME.ShowModal()

END



PUBLIC SUB RunEdit(Data AS Result)

  hResul = Data
  Editando = TRUE

  TxtTitulo.Text = hResul["titulo"]
```

```
  TxtAutor.Text = hResul["autor"]
  TxtFecha.Text = hResul["fecha"]
  TxtPrecio.Text = hResul["precio"]
  TxtDescripcion.Text = hResul["descripcion"]

  ME.ShowModal()

END



PUBLIC SUB BtnCancelar_Click()

  ME.Close()

END

150%

PUBLIC SUB BtnAceptar_Click()

  IF Editando THEN
    TRY hResul["titulo"] = TxtTitulo.Text
    TRY hResul["autor"] = TxtAutor.Text
    TRY hResul["fecha"] = TxtFecha.Text
    TRY hResul["precio"] = TxtPrecio.Text
    TRY hResul["descripcion"] = TxtDescripcion.Text
    TRY hResul.Update()
  ELSE

  TRY hConn.Exec("insert into datos values
  (&1,&2,&3,&4,&5)", TxtTitulo.Text, TxtAutor.Text,
  CDate(TxtFecha.Text), CFloat(TxtPrecio.Text),
  TxtDescripcion.Text)
```

```
    END IF

    ME.Close()

  CATCH
    Message.Error("Imposible introducir los datos
    solicitados")

  END
```


A lo largo de este código, se abre y cierra la conexión con la base de datos en cada operación realizada. Con esto, el periodo de tiempo en que los datos se encuentran en buffers intermedios es menor, reduciendo la posibilidad de corrupciones, ante un fallo de tensión, por ejemplo. No obstante la carga del servidor y cliente trabajando de este modo es mayor, y puede ser un modelo excesivamente lento si el volumen de datos es muy grande, o si sólo se realizan operaciones de lectura. Es conveniente, por tanto, evaluar la conveniencia o no de mantener la conexión y consultas abiertas en las distintas operaciones que se efectúen sobre la base de datos.

5. 5 Otras características

Estructura de las tablas

Gambas también permite examinar la estructura de las tablas de una base. Para ello se emplea la clase **Table** y la clase **Field**. El programador no puede crear objetos de esas clases directamente, pero puede obtenerlos de una conexión en curso. Esta función de ejemplo, lista en la consola la estructura de una tabla, recibe como parámetros una conexión y el nombre de una tabla, y lista sus campos.

```
PRIVATE FUNCTION ListarTabla ( hConn AS Connection,
NombreTabla AS String)
```

```
hTable = hConn.Tables[NombreTabla]

  FOR EACH hField IN hTable.Fields
    PRINT "-------------"
    PRINT "Campo: " & hField.Name
    SELECT CASE hField.Type
      CASE gb.Integer
        PRINT "Entero"
      CASE gb.Float
        PRINT "Número real"
      CASE gb.String
        PRINT "Cadena"
      CASE gb.Date
        PRINT "Fecha/Hora"
      CASE gb.Boolean
        PRINT "Booleano"
    END SELECT
  NEXT

END
```


En ocasiones puede ser necesario examinar una tabla, como al inicio del programa, para comprobar sus campos. Por ejemplo, en la fase de transición moneda nacional/Euro en Europa, se hizo necesario almacenar datos de importes en ambas monedas durante un periodo, en algunos programas. Las versiones actualizadas de estos programas, al instalarse y arrancar por primera vez, comprobaban la existencia de estos campos, y actualizaban la base de forma automática. Versiones posteriores podían eliminar el campo de moneda nacional para ahorrar espacio en la base.

SQL no es sólo un lenguaje de consulta. Las instrucciones como *CREATE TABLE*, *DELETE TABLE* y *ALTER TABLE*, se utilizan precisamente para la función indicada: alterar la estructura de la base y sus tablas cuando es necesario.

Más utilidades del Gestor de Bases de Datos

Ejecutando *gambas-database-manager* podemos encontrar algunas utilidades adicionales.

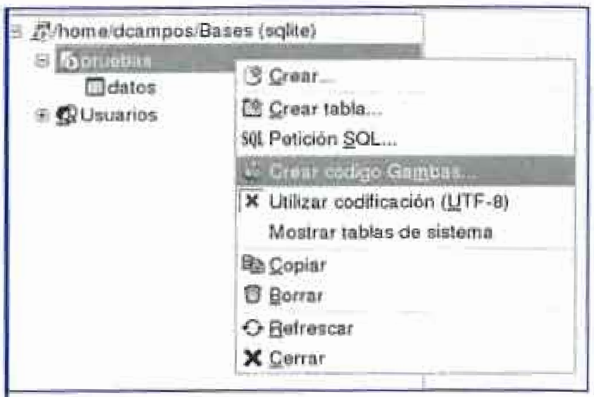


Figura 24. Otras utilidades del Gestor de Bases de Datos.

La primera de ellas es obtener el código para crear las tablas de la base. Situándonos sobre una base y pulsando el botón derecho, podemos seleccionar la opción **Crear código Gambas...**

Al acceder a este menú, nos preguntará el nombre del módulo y el procedimiento para crear el código. Tras indicar estos valores, cerrar y volver a abrir el proyecto, dispondremos de un nuevo módulo con el nombre indicado, que contiene el código para crear la tabla.


```
PROCEDURE CreateDatabase(hConn AS Connection,
sDatabase AS String)

  ' Generated by the Gambas database manager -
22/08/2005 10:26:43

  DIM hTable AS Table

  hTable = hConn.Tables.Add("datos")

  WITH hTable

    .Fields.Add("titulo", gb.String, 40)
    .Fields.Add("autor", gb.String, 40)
    .Fields.Add("fecha", gb.Date)
    .Fields.Add("precio", gb.Float)
```

```
        .Fields.Add("descripcion", gb.String, 200)

        .PrimaryKey = ["titulo"]

        .Update

    END WITH

END
```

Al inicio del programa, y tras conectar con el servidor de bases de datos, podemos entonces comprobar si no existe la(s) tabla(s), y llamar a este código para que la(s) cree, con lo cual se facilita la distribución del programa en varios equipos, sin necesidad de intervención adicional manual para poner en marcha un sistema nuevo.

Gambas-database-manager permite también copiar datos entre bases. Podemos, por ejemplo, pulsar el botón derecho sobre nuestra tabla *datos*, presionar sobre **Copiar**, acceder a otra base de datos, sea Sqlite o de otro tipo, pulsar **Pegar**, y el gestor creará la tabla en la otra base, preguntándonos si queremos copiar sólo la estructura de los campos o también los datos. De esta forma, se facilita la fase de programación, al poder disponer de varias bases de pruebas, así como la fase de migración de una base a otra, al poder copiar datos de la base antigua a la nueva. Este proceso se puede llevar a cabo para tablas individuales o para una base de datos completa.

6. Red

6.1 Conceptos

La transmisión de datos a través de una red se estandarizó hace tiempo a nivel de software y hardware. Actualmente se trabaja con el llamado *modelo de capas*. Básicamente, este modelo pretende aislar los diferentes niveles de complejidad de datos, de forma que la sustitución de una capa no afecte al trabajo en otra. Podemos ver hoy en día que es posible, por ejemplo, conectarse a Internet a través de un módem, una línea ADSL o cable ethernet en una red local y, sin embargo, el hecho de tener aparatos tan distintos en funcionamiento no obliga a disponer de un navegador o un cliente de correo diferentes para cada situación. Esto se debe al modelo de capas. El navegador sólo entiende de protocolos de alto nivel, como HTTP o FTP, transfiere la información a un nivel más bajo y, finalmente, el módem convencional, ADSL o la tarjeta de red se ocupan de transportar esos datos según proceda.

Para no entrar en complejidades, ya que es tarea de un libro sobre redes, diremos que hay un par de niveles: los más bajos, conformados por el hardware (módems, tarjetas de red...) y los drivers o módulos del núcleo, que controlan y trocean la información para emitirla o recibirla por dicho hardware; y unos niveles por encima, que son los que más nos interesan a la hora de programar con Gambas aplicaciones de red, y que son independientes del método de transporte que haya por debajo.

El protocolo más extendido hoy en día para distribuir información entre equipos es el protocolo IP, que determina qué destino y qué equipo recibirá cada fragmento de información que circula por una red. Cada equipo tiene un número asignado, llamado dirección IP es su identificador único. Cada paquete IP de información es similar a una carta postal: incluye datos del remitente (IP del equipo de origen) y del destinatario (IP del equipo de destino). Estas direcciones IP tienen la forma *XXX.XXX.XXX.XXX*; son grupos de cuatro números que varían entre 0 y 255, separados por puntos (por ejemplo: 192.168.0.23 o 10.124.35.127).

Dado que estos números son difíciles de recordar, se estableció un sistema que permitía establecer una correspondencia entre nombres y direcciones IP. Este sistema se denomina DNS y, a lo largo y ancho de Internet y de casi todas las redes locales, se encuentran servidores DNS que reciben consultas acerca de nombres de equipos y direcciones IP y las traducen en un sentido u otro. De esta forma, podemos, por ejemplo, indicar al navegador que muestre la página **www.gnulinex.org**, en lugar de tener que indicar la dirección IP del equipo que está sirviendo esta página.

Los paquetes IP pueden contener cualquier tipo de información, pero encima de este nivel se han establecido otros dos también estándar, que son los protocolos TCP y UDP.

1. **TCP** se utiliza para asegurar la conexión entre dos equipos: se sabe en todo momento si el equipo remoto está a la escucha, si ha recibido toda la información correctamente o hay que volverla a emitir, y se añaden sistemas de control de errores para asegurar que un ruido en la línea no ha deformado los mensajes. Es el protocolo más utilizado en Internet, ya que asegura que recibamos una página web o un fichero de forma completa y sin errores.

2. En cuanto a UDP, es un protocolo de transporte mucho más simple que TCP, y no verifica si realmente existe una conexión entre los dos equipos o si la información ha sido realmente recibida o no. Es, por tanto, menos fiable, pero en determinados tipos de transmisiones, como son las de audio y vídeo en tiempo real, lo importante no es que llegue toda la información (puede haber pequeños cortes o errores), sino que el caudal sea constante y lo más rápido posible. Por ello, también se emplea con frecuencia en Internet, sobre todo para los llamados servidores de *streaming*, que nos permiten escuchar radio, ver programas de televisión o realizar videoconferencias por la red.

A la hora de establecer una comunicación entre dos equipos, el modelo indica que ha de abrirse un *socket*. Es algo similar a una bandeja de entrada o salida, como la de los administrativos, que tienen una bandeja con los informes pendientes y otra con los que van terminando. El sistema operativo almacena información en la bandeja de entrada proveniente del sistema remoto, hasta que nuestro programa decide tomar los datos, entonces los procesamos y los dejamos en la bandeja de salida. El sistema operativo se encargará de enviar esa información cuando le sea posible.

Sockets no se emplea sólo para comunicar dos o más equipos, dentro de nuestra propia máquina muchos programas se comunican entre sí utilizando este sistema. Por ejemplo, es el caso de los servidores gráficos o X-Window: el servidor gráfico dibuja lo que le piden los programas clientes o aplicaciones que han establecido un socket con él.

Existe un tipo especial de socket, llamado *local* o *Unix*, que sólo sirve para comunicar programas dentro de un mismo sistema, y que está optimizado para realizar esa función con gran velocidad y eficacia, permitiendo una comunicación interna varias veces más veloz y con menor consumo de recursos que a través de un socket TCP o UDP.

Subiendo más arriba, tenemos los llamados *protocolos de aplicación*. Ya no nos encargamos del transporte de datos, sino del formado de los datos y de la comunicación. Uno de los protocolos más extendidos es el HTTP que, entre otras cosas, se utiliza

en todo Internet para transmitir y recibir páginas web. Establece el modo de conectar, cómo autenticarse, el formato de los datos a transmitir o recibir (páginas web, imágenes, archivos comprimidos...) y cómo finalizar la comunicación.

Otros protocolos específicos son el FTP, especializado en la transmisión y recepción de archivos; TELNET, para trabajar con un terminal de texto sobre un sistema remoto; SMTP, POP o IMAP para el correo electrónico o los diferentes protocolos de mensajería instantánea, como el JABBER.

Si esta breve introducción nos ha resultado nueva, debemos extender nuestros conocimientos en el área de redes, aprendiendo conceptos como las máscaras de red y divisiones en subredes, los diferentes tipos de proxies, routers y gateways, las conversiones NAT, el futuro protocolo Ipv6 y, en general, todo lo que nos ayude a determinar por qué no conseguimos conectar con otro equipo en un momento dado. El mundo de las redes es extenso, pero una buena base nos evitará muchos problemas a la hora de programar sistemas distribuidos entre varios servidores y clientes.

6. 2 Creando un servidor TCP

Nuestra primera tarea consistirá en escribir el código de un servidor TCP: será un servidor que aceptará conexiones remotas, leerá los datos que envían y devolverá un eco a los clientes. Crearemos un programa de consola llamado *MiServidor*. Este programa tendrá un único módulo llamado **ModMain**. En las propiedades del proyecto hemos de seleccionar el componente **gb.net**.

Dentro del módulo **ModMain** tendremos una referencia a un objeto *ServerSocket*. Dichos objetos se comportan como servidores de sockets, es decir, se encuentran a la escucha de peticiones de conexión de clientes remotos o locales en un puerto determinado. Los puertos se numeran del 1 al 65535 y cada programa que actúa como servidor dentro del sistema utiliza uno de ellos.

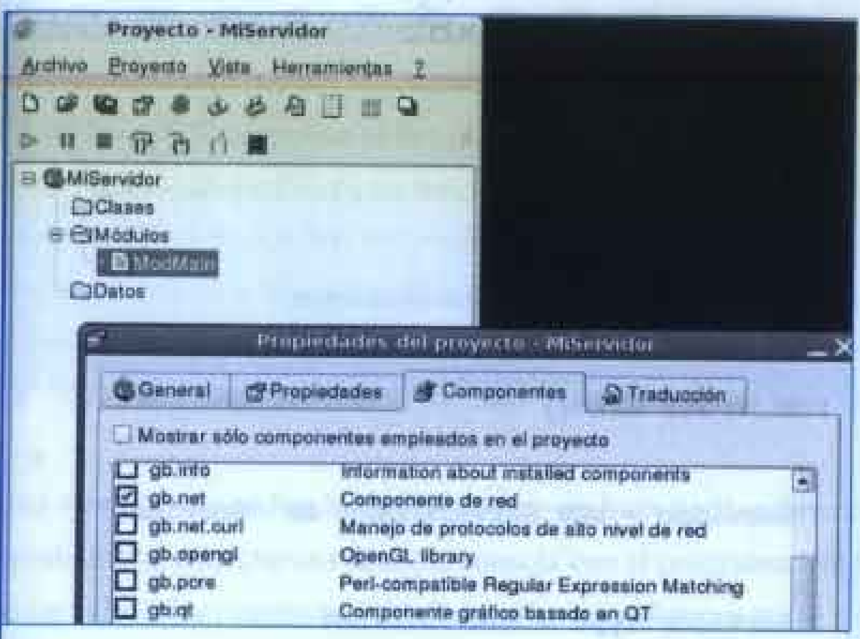


Figura 1. Programa *MiServidor*.

Los puertos con número del 1 al 1024, se consideran reservados para servicios conocidos (es decir, HTTP, POP, FTP, IMAP...) y no pueden ser utilizados por programas cuyo usuario es distinto del root en sistemas GNU/Linux. Tratar, por tanto, de abrir, por ejemplo, el puerto 523 desde un programa ejecutado por un usuario normal, dará lugar a un error.

Los servicios más conocidos como HTTP o FTP ya tienen un puerto asignado (por ejemplo, 80 en el caso de HTTP) de forma estándar, si bien no hay ninguna razón técnica por la que un servidor web no pueda atender, por ejemplo, al puerto 9854. Esto se debe únicamente a un acuerdo internacional para que cualquier cliente que quiera realizar una petición web, por ejemplo, sepa que, salvo instrucciones específicas en otro sentido, ha de conectarse al puerto 80 de la máquina servidora.

En sistemas GNU/Linux podemos encontrar una lista de los servicios más comunes y sus puertos *oficiales* dentro del fichero de texto */etc/services*.

En nuestro caso, vamos a emplear el puerto 3152. En la función Main del programa, que se ejecuta al inicio de éste, habremos de crear el objeto ServerSocket, especificar que su tipo será **Internet** (es decir, que es un socket TCP y no UNIX), el puerto al que debe atender e indicarle que comience con las peticiones.

```
PRIVATE Servidor AS ServerSocket

PUBLIC SUB Main()

  Servidor = NEW ServerSocket AS "Servidor"

  Servidor.Type = Net.Internet
  Servidor.Port = 3152
  TRY Servidor.Listen()
  IF ERROR THEN PRINT "Error: el sistema no permite
  atender al puerto especificado"

END
```


Observemos que a la hora de crear el objeto servidor, indicamos que el gestor de eventos será **"Servidor"**, para poder escribir las rutinas en las cuales tratemos las peticiones que llegan de los clientes.

A la hora de indicar al servidor que se ponga a la escucha con el método Listen, lo hacemos protegiendo el código con una instrucción TRY, para gestionar el error si el sistema no permitiese atender al puerto indicado (por ejemplo, si otro servicio ya lo estuviera utilizando). Ya podemos ejecutar el programa.

Como podemos observar, el programa pasa la función Main y sigue ejecutándose a la espera de una conexión de un cliente. Un programa normal de consola, al terminar esta función, hubiera finalizado sin más, pero en este caso el intérprete Gambas está atendiendo las novedades que puedan acaecer en el socket que se ha creado y, por tanto, el programa sigue funcionando, esperando peticiones de clientes.

Cualquier programa Gambas que está vigilando un *descriptor*, es decir, un archivo, un proceso o un socket, no finaliza mientras dicho *descriptor* se encuentre abierto. Esta técnica permite crear programas de consola que no se encuentren en un bucle continuo a la espera de novedades (sea una modificación en un fichero, una entrada de un cliente en un socket o la lectura de datos de un proceso hijo), ahorrando recursos y mejorando la eficacia del proceso.

El programa servidor ya está funcionando, aunque no haga nada realmente útil. De hecho, si probamos a conectarnos desde la consola con el programa telnet a nuestro propio servidor, observaremos que se conecta e inmediatamente se desconecta del servicio.

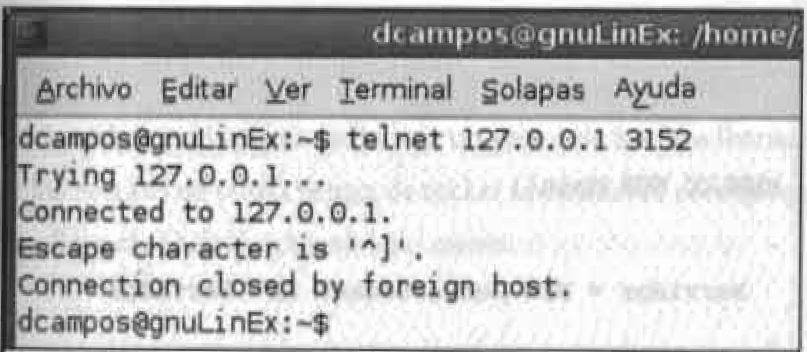


Figura 2. Prueba de conexión.

El funcionamiento del objeto ServerSocket es precisamente éste: recibe una conexión de un cliente y, si no especificamos en el programa que deseamos atenderla, la cierra.

Cuando un objeto servidor recibe una petición de un cliente, se dispara el evento **Connection**. Dentro de dicho evento, y sólo dentro de él, podemos utilizar el método **Accept**. Dicho método devuelve un objeto **Socket** que representa una conexión con dicho cliente. A la hora de recibir o enviar datos a ese cliente, se hará a través de dicho objeto, de ese modo, se diferencia a cada cliente conectado con un servidor de forma única, evitando, por ejemplo, que enviemos los resultados de un cálculo a un cliente equivocado.

En nuestro programa añadiremos una matriz de objetos donde guardaremos una referencia a cada objeto cliente, y en esa matriz iremos añadiendo los clientes que se conectan.

```
PRIVATE Servidor AS ServerSocket
PRIVATE Clientes AS NEW Object[]

PUBLIC SUB Servidor_Connection(RemoteHostIP AS String)

  DIM Cliente AS Socket

  Cliente = Servidor.Accept()
  Clientes.Add(Cliente)

END


PUBLIC SUB Main()

  Servidor = NEW ServerSocket AS "Servidor"

  Servidor.Type = Net.Internet
  Servidor.Port = 3152
  TRY Servidor.Listen()
  IF ERROR THEN PRINT "Error: el sistema no permite
  atender al puerto especificado"

END
```

En nuestro gestor de eventos Connection utilizamos el método Accept, por lo que siempre aceptamos las conexiones entrantes. No obstante, si lo omitimos en casos determinados, el intento de conexión del cliente se cerrará automáticamente. Observamos, por ejemplo, que Accept nos informa de la dirección IP del cliente que

trata de conectarse a través del parámetro **RemoteHostIP**. Si deseáramos, por ejemplo, aceptar sólo conexiones desde nuestro propio equipo (que siempre tiene dirección 127.0.0.1), podríamos modificar el código para que se rechacen las conexiones desde otras direcciones IP.

```
PUBLIC SUB Servidor_Connection(RemoteHostIP AS String)

  DIM Cliente AS Socket

  IF RemoteHostIP="127.0.0.1" THEN
    Cliente = Servidor.Accept()
    Clientes.Add(Cliente)
  END IF

END
```

Estos objetos socket, por defecto reciben ya un gestor de eventos llamado **Socket**, por lo que los eventos de estos clientes de socket se atenderán en el programa por una función llamada *Socket_ + Nombre del evento.*

En nuestro caso atenderemos a los eventos *Read* que se producen cuando se reciben datos desde el cliente, y el evento *Close*, que se produce cuando el cliente cierra la conexión.

En el caso del evento READ, nos valdremos de la palabra clave LAST, que mantiene una referencia al último objeto que ha disparado un evento (en este caso, el cliente de socket) para tratar los datos que provienen de él.

Para ello, leeremos el socket como si fuera un archivo, con la instrucción READ, en la que indicamos una cadena que recibe los datos; leeremos la longitud de los datos disponibles en el socket, determinada por la función Lof() y, después, escribiremos esos mismos datos en el socket con la instrucción WRITE, de forma que el cliente reciba un eco de los datos enviados, pero convertidos a mayúsculas.

```
PUBLIC SUB Socket_Read()

  DIM sCad AS String

  TRY READ #LAST, sCad, Lof(LAST)
  sCad=UCase(sCad)
  TRY WRITE #LAST, sCad, Len(sCad)

END
```

Para Close, buscamos el objeto Socket dentro de nuestra matriz, y lo eliminamos para que desaparezcan las referencias a dicho cliente dentro de nuestro programa servidor.

```
PUBLIC SUB Socket_Close()

  DIM Ind AS Integer

  Ind = Clientes.Find(LAST)
  IF Ind >= 0 THEN Clientes.Remove(Ind)

END
```

Ya disponemos de un programa servidor de socket con capacidad para atender a múltiples clientes: podemos abrir varias ventanas de terminal, ejecutando en ellas **telnet 127.0.0.1 3152**, y enviar y recibir datos del servidor. Cada vez que escribamos una cadena y la enviemos pulsando **Return**, recibiremos la cadena convertida a mayúsculas como respuesta del servidor.

> Aunque el programa telnet se diseñó para controlar un equipo de forma remota, también es un *cliente universal* que puede servir para comprobar el funcionamiento de cualquier servidor que estemos diseñando a mano, antes de disponer de un cliente real.

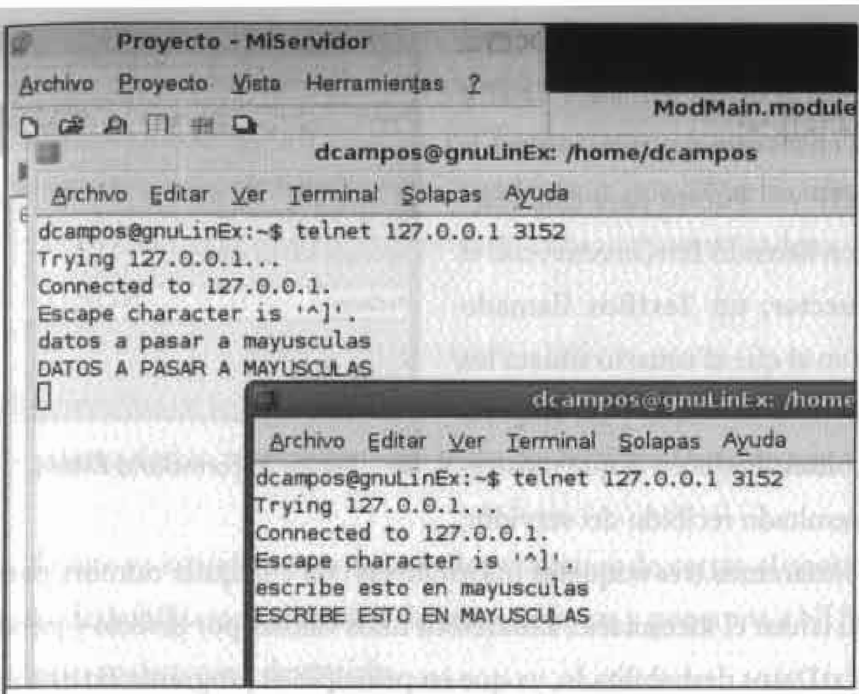


Figura 3. Programa servidor de socket.

6. 3 Un cliente TCP

Ahora que ya disponemos de un servidor, vamos a crear un programa cliente.

Diseñaremos un programa gráfico llamado *MiCliente*, con un formulario FMain, y una referencia al componente **gb.net** como en el caso anterior.

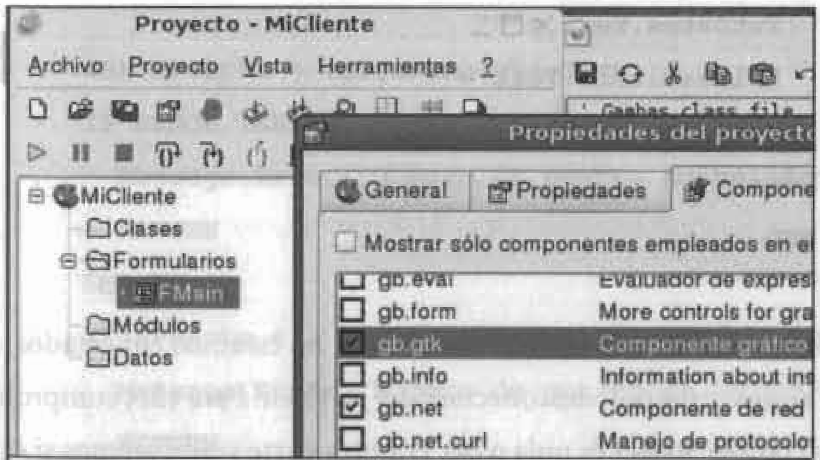


Figura 4. Proyecto *MiCliente*.

Dentro del formulario FMain situaremos dos controles TextBox, llamados *TxtIP* y *TxtPuerto*, en los que el usuario situará la Dirección IP y el Puerto para conectarse; un botón llamado *BtnConectar*, con el texto Conectar; un TextBox llamado *TxtDatos*, en el que el usuario situará los datos a enviar al servidor, y un etiqueta llamada **LblResultado**, en la que mostraremos el resultado recibido del servidor.

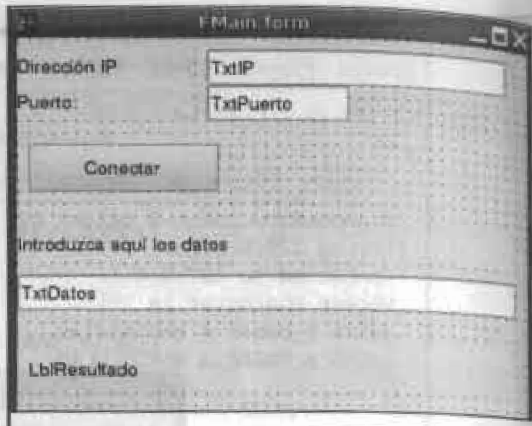


Figura 5. Formulario *FMain*.

También situaremos tres etiquetas informativas con cualquier nombre para guiar al usuario. Al lanzar el formulario, situaremos unos valores por defecto y pondremos el TextBox TxtDatos deshabilitado, ya que en principio el programa no está conectado al servidor. También situaremos al principio del código una variable global de tipo Socket, que representará al cliente con el que nos conectaremos al servidor.

```
' Gambas class file
PRIVATE Cliente AS Socket

PUBLIC SUB Form_Open()

  TxtIp.Text = "127.0.0.1"
  TxtPuerto.Text = "3152"
  TxtDatos.Text = ""
  LblResultado.Text = ""
  TxtDatos.Enabled = FALSE

END
```

El botón BtnConectar tendrá dos funciones: si no estamos conectados, tratará de conectar, y si ya lo estamos, desconectará del servidor. Para ello, comprobamos si la referencia al objeto Socket es nula o no. Por otra parte verificaremos si el dato de la dirección IP es válido, así como el del puerto indicado.

Para verificar el número de puerto usamos dos funciones: Val(), devuelve un número a partir de un texto si éste contiene sólo caracteres numéricos. Si no es así, devuelve NULL. Si hemos obtenido un número, verificamos que se encuentre en el rango 1-65535. Para la dirección IP, emplearemos Net.Format, que devuelve una dirección IP a partir de un texto, si es posible interpretarlo como tal, o una cadena vacía si no es una dirección IP.

Una vez disponemos de los datos, creamos el objeto **Socket** y tratamos de conectar con la IP y puerto dados, y cambiamos el texto del botón a *Desconectar*.

En el caso de que ya estuviéramos conectados tratamos de cerrar el socket, si estuviera abierto, deshabilitamos el cuadro de texto de datos y ponemos a NULL la referencia al objeto socket, para destruirlo.

```
PUBLIC SUB BtnConectar_Click()

  DIM nPuerto AS Integer
  DIM sIP AS String

  LblResultado.Text = ""
  TxtDatos.Text = ""

  IF Cliente = NULL THEN

    TRY nPuerto = Val(TxtPuerto.Text)
    IF ERROR THEN
      Message.Error("Número de puerto no válido")
      RETURN
    END IF
    IF nPuerto < 1 OR nPuerto > 65535 THEN
      Message.Error("Número de puerto no válido")
      RETURN
    END IF
```

```
    sIP = Net.Format(TxtIP.Text)
    IF sIP = "" THEN
      Message.Error("Dirección IP no válida")
      RETURN
    END IF

    Cliente = NEW Socket AS "Cliente"
    Cliente.Host = sIP
    Cliente.Port = nPuerto
    Cliente.Connect()

  ELSE

    TRY CLOSE #Cliente
    Cliente = NULL
    TxtDatos.Enabled = FALSE

  END IF

END
```

Entre el intento de conexión y la conexión real puede pasar un intervalo de tiempo. Para saber cuándo hemos conectado realmente con el servidor, emplearemos el evento Ready, que se produce cuando el servidor acepta nuestra conexión. En este evento, habilitaremos el cuadro de texto de datos para que el usuario pueda escribir allí. También puede suceder que se produzca un error (por ejemplo, una conexión rechazada o un nombre de host no encontrado), en cuyo caso se dispara el evento Error, que aprovechamos para devolver la interfaz a su estado desconectado y eliminar el socket fallido.

```
PUBLIC SUB Cliente_Ready()

  TxtDatos.Enabled = TRUE
```

```
END

PUBLIC SUB Cliente_Error()

  TRY CLOSE #Cliente
  Cliente = NULL
  TxtDatos.Enabled = FALSE
  BtnConectar.Text = "Conectar"
  Message.Error ("Conexión cerrada")

END
```

En cuanto al envío de datos, lo realizaremos cuando el usuario pulse la tecla **Return**, tras escribir un texto en el cuadro TxtDatos.

Para ello introduciremos el código dentro del evento **KeyPress** del cuadro de texto.

```
PUBLIC SUB TxtDatos_KeyPress()

  IF Key.Code = Key.Return THEN

    IF Len(TxtDatos.Text) > 0 THEN

      LblResultado.Text = ""
      TRY WRITE #Cliente, TxtDatos.Text,
      Len(TxtDatos.Text)

    END IF

  END IF

END
```

En cuanto a la recepción de datos del servidor, puede llegarnos en varios fragmentos, cada uno de los cuales recibiremos en el evento Read del objeto socket, y empalmaremos en la etiqueta **LblResultado**.

Cuando se envían y reciben datos en una red, nunca se puede asegurar qué cantidad de datos se reciben de una sola vez, siempre hay que tener en cuenta la posibilidad de unir fragmentos de los datos totales.

```
PUBLIC SUB Cliente_Read()

  DIM sCad AS String

  TRY READ #Cliente, sCad, Lof(Cliente)
  LblResultado.Text = LblResultado.Text & sCad

END
```


Ya tenemos nuestro cliente listo. Arrancamos el servidor que creamos anteriormente y una o varias instancias de este programa para comprobar los resultados.

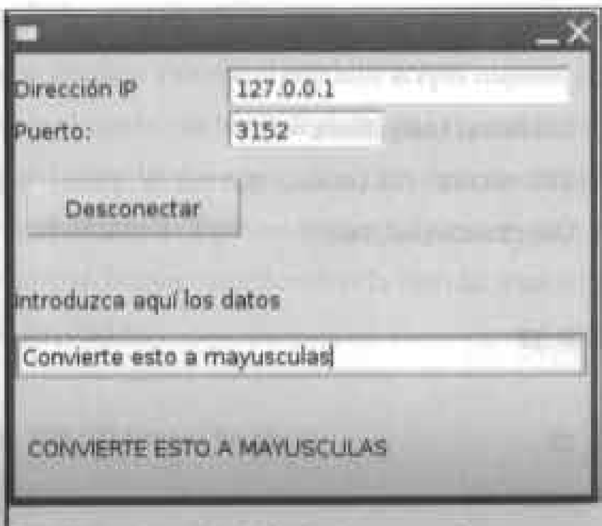


Figura 6. Resultado final de nuestro cliente.

A la hora de crear clientes o servidores para producción, debemos tener en cuenta la codificación de los caracteres empleada por el servidor y el cliente. Muchos errores pueden provenir de no tener en cuenta, simplemente, que Gambas y muchos otros programas usan internamente UTF-8 como codificación predeterminada, mientras que otros emplean ISO-8859-1 o UTF-16. Este programa, por ejemplo, puede tener problemas con la *ñ* y vocales acentuadas. Podemos mejorarlo como ejercicio, convirtiendo las cadenas enviadas y recibidas entre UTF-8 e ISO-8859-1 o ISO-8859-15.

6. 4 Clientes y servidores locales

Al margen de los sockets TCP, diseñados para conectar equipos remotos, los sistemas GNU/Linux, y en general cualquier sistema que siga la filosofía de la familia de sistemas UNIX™, disponen de otro tipo de sockets, que sólo permiten la conexión dentro del propio equipo y cuya misión es optimizar la funcionalidad de los sockets cuando el cliente y servidor se encuentran en la misma máquina.

Estos sockets son unos ficheros especiales que el servidor crea dentro del sistema de archivos, y que los clientes tratan de abrir para lectura y escritura de datos. Hemos de tener en cuenta que situar uno de estos ficheros especiales dentro de una unidad de red no sirve para conectar dos equipos diferentes: un socket UNIX o local, sólo se puede emplear dentro de un mismo equipo.

Todo lo explicado hasta aquí para clientes y servidores TCP, es aplicable a los clientes y servidores locales, la única diferencia se da en el modo inicial de configurar el servicio.

En el caso del servidor, antes de conectar hemos de especificar que el tipo de servidor es local y una ruta a un archivo que representará el socket servidor.

```
' Servidor TCP:
PUBLIC SUB Main()
```

```
Servidor = NEW ServerSocket AS "Servidor"

Servidor.Type = Net.Internet
Servidor.Port = 3152
TRY Servidor.Listen()
IF ERROR THEN PRINT "Error: el sistema no permite
atender al puerto especificado"

END

' Servidor "Local" o "UNIX"
PUBLIC SUB Main()

Servidor = NEW ServerSocket AS "Servidor"

Servidor.Type = Net.Local
Servidor.Path = User.Home & "/misocket"
TRY Servidor.Listen()
IF ERROR THEN PRINT "Error: el sistema no permite
atender al puerto especificado"

END
```

Si ejecutamos el código con esta modificación, observaremos que en nuestra carpeta personal se crea un archivo especial de tipo socket llamado misocket. En la parte cliente hemos de especificar que la conexión se realiza con una ruta dentro del sistema de archivos, en lugar de un puerto TCP, para lo cual se indica el puerto especial Net.Local y la propiedad Path del socket.

```
Cliente = NEW Socket AS "Cliente"
Cliente.Path = User.Home & "/misocket"
Cliente.Port = Net.Local
Cliente.Connect()
```

A partir de aquí el resto del código funcionará sin modificaciones.

Es una buena idea plantear la posibilidad de que servidores y clientes funcionen con sockets locales o TCP, según los parámetros de configuración de la aplicación, con el fin de optimizar el funcionamiento cuando cliente y servidor se encuentran en la misma máquina. Los servidores X y los de fuentes gráficas emplean este modo de trabajo.

6.5 UDP

El trabajo con UDP al principio puede resultar algo más complejo al principiante. En UDP no existe una conexión entre cliente y servidor, tan sólo llegan datos por un puerto, o se envían, pero sin conocer nunca si al otro lado hay alguien a la escucha. De hecho, no existen servidores o clientes realmente, aunque en la práctica sí haremos esta distinción a la hora de diseñar una aplicación servidora o cliente. Puesto que no existe conexión, cada paquete se identifica con su dirección IP y puerto de origen, y este dato ha de ser tenido en cuenta siempre a la hora de devolver una respuesta al lado remoto.

Gambas da acceso al protocolo UDP a través de los objetos de la clase **UdpSocket**, que sirven tanto para crear programas servidores como programas cliente. Para iniciar el trabajo con un objeto de esta clase, se ha de llamar al método *Bind*, en el cual se ha de indicar el puerto local al que estará ligado. Los servidores habrán de estar unidos a un puerto concreto definido, y los clientes a cualquier puerto, ya que el interés del cliente es enviar datos a un servidor remoto en un puerto dado, independientemente del puerto local que emplee para ello.

UdpSocket proporciona varias propiedades para identificar los paquetes. Cuando llega un paquete al socket procedente de una ubicación remota, se genera un evento *Read* y en él podemos consultar la IP de procedencia, con la propiedad **SourceHost**, y el puerto de origen, con la propiedad **SourcePort**. Para enviar un paquete a un sistema remoto, hemos de rellenar previamente el valor de las propiedades **TargetHost**, con la IP de destino, y **TargetPort**, con el puerto de destino.

Crearemos un servidor muy sencillo: atendemos al puerto UDP 3319, recibe los datos y envía como respuesta el texto ADIOS al host que envió el paquete. Para implementarlo crearemos un proyecto de consola llamado **MiServidorUDP**, con un módulo *ModMain* y una referencia al componente **gb.net**. El código es el siguiente:

```
' Gambas module file
PRIVATE MiServidor AS UdpSocket

PUBLIC SUB Servidor_Read()

  DIM sCad AS String

  READ #MiServidor, sCad, Lof(MiServidor)

  MiServidor.TargetHost = MiServidor.SourceHost
  MiServidor.TargetPort = MiServidor.SourcePort
  WRITE #MiServidor, "ADIOS", 5

END


PUBLIC SUB Main()

  MiServidor = NEW UdpSocket AS "Servidor"
  MiServidor.Bind(3319)

END
```

Dispondremos de un objeto de la clase **UdpSocket** que creamos en la función **Main**. Hacemos una llamada a **Bind** para enlazarlo con el puerto UDP 3319. Como en el caso del servidor TCP, el intérprete Gambas queda a la espera de algún evento en el socket. Cuando llega un paquete de datos, se lee como en el caso de los sockets TCP, haciendo uso de la instrucción **READ**, a continuación situamos las propiedades

TargetHost y TargetPort con los valores correspondientes a la procedencia del paquete de datos, y enviamos la cadena ADIOS, que llegará al host especificado. En cuanto a la parte cliente, se tratará de otro programa de consola que llamaremos MiClienteUDP, con referencia al componente *gb.net*, y que tendrá un único módulo llamado *ModMain*. El cliente tratará de conectar con el servidor, enviará una cadena HOLA y cuando reciba una cadena completa ADIOS, cerrará el socket y acabará su funcionamiento.

```
' Gambas module file
PRIVATE MiCliente AS UdpSocket
PRIVATE Buffer AS String

PUBLIC SUB Cliente_Read()

  DIM sBuf AS String

  READ #MiCliente, sBuf, Lof(MiCliente)
  Buffer = Buffer & sBuf
  IF Buffer = "ADIOS" THEN CLOSE #MiCliente

END

PUBLIC SUB Main()

  MiCliente = NEW Udpsocket AS "Cliente"

  MiCliente.Bind(0)

  MiCliente.TargetHost = "127.0.0.1"
  MiCliente.TargetPort = 3319
  WRITE #MiCliente, "HOLA", 4

END
```

En este caso, se hace también al inicio una llamada a **Bind**, pero asignando el valor especial **0** para indicar que el sistema elija por nosotros cualquier puerto local que se encuentre libre. Tras esto, rellenamos los datos de **TargetHost** y **TargetPort** con los datos de destino del paquete y enviamos la cadena **HOLA**.

En el evento **READ**, leemos los fragmentos de datos procedentes del servidor, los unimos en la cadena **Buffer** y cuando ésta toma el valor **ADIOS** cerramos el socket. Al cerrarlo, el intérprete Gambas deja de aguardar eventos y finaliza normalmente.

6. 6 Resolución de nombres

El componente de red también aporta una clase *DnsClient* para obtener la correspondencia entre direcciones IP y nombres de equipo. Si bien el nombre sugiere la llamada a servicios DNS de resolución, lo cierto es que los objetos de esta clase tratarán de resolver el nombre por todos los medios que disponga el sistema. Lo más común es que en las redes de área local se emplee el fichero */etc/hosts* que contiene una lista de pares nombre/IP, servidores DNS, LDAP o NIS; mientras que en accesos a Internet se emplee, normalmente, sólo el servicio DNS. El funcionamiento de esta clase es muy sencillo. Mediante el método *GetHostIP* se trata de obtener la IP de un nombre de equipo (*Host*, *Name*), y mediante el método *GetHostName* realizar una resolución inversa, es decir, de dirección IP a nombre de equipo.

Crearemos un proyecto gráfico llamado MiClienteDNS con una referencia al componente *gb.net*, así como un formulario llamado *Fmain*. Dentro de él situaremos un control TextBox llamado *TxtHost*, con su propiedad *Text* inicialmente en blanco y dos botones, uno llamado BtnToIp con el texto **Obtener IP**, y otro llamado BtntoName con el texto **Obtener nombre**. En esta ocasión, en lugar de crear un objeto **DnsClient** en ejecución, vamos a añadir el control **DnsClient** de la lista de controles disponibles en el diseño del formulario, dentro de la pestaña **Network**.

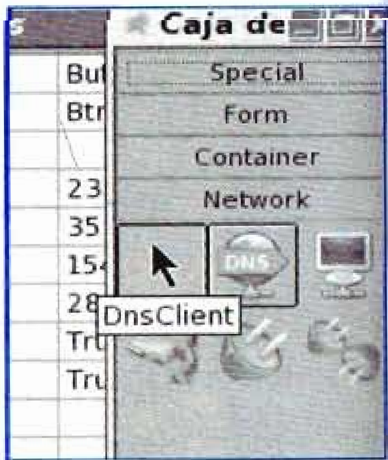


Figura 7. Proyecto *MiClienteDNS*.

El control DnsClient recibirá el nombre *Cliente* en nuestro formulario, y la propiedad *Async* del cliente DNS se debe situar con el valor *FALSE*.

Estos controles especiales (DnsClient, Socket, etc.) no son visibles en tiempo de ejecución, ni son realmente controles, pero se añaden al formulario como controles normales con el fin de facilitar la labor al crear programas gráficos. El objeto DnsClient, en nuestro caso llamado *Cliente*, se crea al desarrollar el formulario y existe hasta que se destruye al cerrarlo o al aplicar el método *Delete* en dicho formulario.

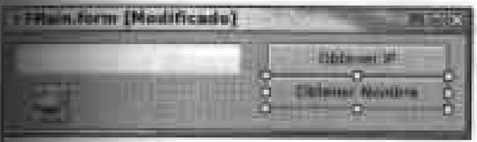


Figura 8. Formulario *FMain*.

El aspecto del diseño será similar al de la Figura 8.

El código será el siguiente:

```
PUBLIC SUB BtnToIp_Click()

  Cliente.HostName = TxtHost.Text
  Cliente.GetHostIP()
  Message.Info("IP: " & Cliente.HostIP)

END

PUBLIC SUB BtnToName_Click()

  Cliente.HostIp = TxtHost.Text
  Cliente.GetHostName()
  Message.Info("Nombre: " & Cliente.HostName)

END
```

En el primer caso, se rellena la propiedad **HostName** con el nombre del host que deseamos resolver; se llama al método **GetHostIP()** y, después, leemos la IP de dicho host recién obtenida con el valor de la propiedad **HostIP**.

El segundo caso es exactamente el contrario: rellenamos la propiedad **HostIp** con la IP que ha escrito el usuario en la caja de texto; llamamos al método **GetHostName()** y leemos y mostramos el valor de la propiedad **HostName** que hemos resuelto.

En ambos casos, si la resolución falla, la propiedad a leer (**HostIP** en el primer caso y **HostName** en el segundo) quedará en blanco. También podemos controlar el estado de error con la propiedad *Status*. Si vale cero, la resolución terminó correctamente; si tiene un valor menor que cero, hubo un error, no se encontró la IP o nombre de host.

DnsClient también permite el trabajo en modo asíncrono. En ocasiones, una resolución puede tardar bastante tiempo, en el que la interfaz de usuario queda bloqueada trabajando en modo síncrono.

Para activar el modo asíncrono, se debe poner la propiedad **Async** del cliente DNS a **TRUE**. Esto también requiere modificar el código. Ya no podemos llamar al método **GetHostIP()** o **GetHostName()** e inmediatamente leer el valor de las propiedades **HostName** o **HostIp** respectivamente, ya que mientras el código del programa principal se está ejecutando, el cliente DNS está trabajando internamente. Tenemos dos opciones: la primera es esperar el evento *Finished*, que se dispara al acabar el proceso; y la segunda es comprobar el estado de la propiedad **Status**, que valdrá un número mayor que cero mientras el proceso se haya en curso, cero cuando finaliza con éxito o un valor menor que cero si hubo un error.

Éste es un pequeño ejemplo modificando el código inicial para trabajar en modo asíncrono. Se comprueba en un bucle el estado de la propiedad **Status**, repitiéndose mientras es mayor que cero. En ese bucle se podría, por ejemplo, controlar la pulsación de un botón **Cancelar** para abortar el proceso.

```
PUBLIC SUB BtnToIp_Click()

  Cliente.Async = TRUE
  Cliente.HostName = TxtHost.Text
  Cliente.GetHostIP()

  DO WHILE Cliente.Status > 0

    WAIT 0.01

  LOOP

  Message.Info("IP: " & Cliente.HostIP)

END

PUBLIC SUB BtnToName_Click()

  Cliente.Async = TRUE
  Cliente.HostIp = TxtHost.Text
  Cliente.GetHostName()

  DO WHILE Cliente.Status > 0

    WAIT 0.01

  LOOP

  Message.Info("Nombre: " & Cliente.HostName)

END
```

6. 7 Protocolo HTTP

En un nivel por encima de todo lo explicado hasta ahora se encuentra el protocolo HTTP, en el que ya no se trata sólo de conectar con un servidor, sino de establecer un formato para la comunicación entre clientes y servidores. El protocolo HTTP es uno de los más extendidos a lo largo de Internet, dado que, entre otras cosas, se emplea para transmitir páginas web.

En el protocolo HTTP, el cliente solicita al servidor un documento en una ubicación dada, y éste lo devuelve siguiendo una serie de convenciones acerca de la codificación de caracteres, control de errores, nivel de compresión de datos, formato de los datos binarios, etc.

El protocolo HTTP establece dos métodos principales para solicitar datos al servidor. El primero, y más común por ser utilizado para recibir páginas web en el navegador cuando escribimos una dirección, por ejemplo, es el llamado método *GET*, en el cual el cliente tan sólo solicita una dirección URL, devolviendo el servidor el resultado. En el segundo método, llamado *POST*, el cliente no sólo solicita la URL, si no que envía una serie de informaciones adicionales que el servidor procesará antes de enviar el resultado. Es el caso habitual cuando rellenamos un formulario con datos en una página web y pulsamos el botón **enviar**.

Gambas aporta en el componente *gb.net.curl*, un cliente llamado *HttpClient*, que provee acceso a servidores HTTP. El trabajo de negociación entre cliente y servidor, así como la gestión de formatos, es tarea interna del cliente HTTP, la aplicación que desarrollemos sólo habrá de preocuparse en pedir un documento y recibirlo.

El cliente HTTP puede funcionar de dos modos: el más simple es el modo síncrono, en el que recibiremos directamente el documento tras la llamada a los métodos *GET* o *POST*; en el segundo, asíncrono, el modo de trabajo se parece más al descrito para los sockets: con la aplicación en funcionamiento iremos recibiendo fragmentos del documento en cada evento *READ* que uniremos en una cadena.

Crearemos un proyecto de consola para recibir la página web **http://gambas.gnulinex.org/gtk/**. Tendrá referencias a *gb.net.curl* y *gb.net*, y un módulo *ModMain*, con este código:

```
PUBLIC SUB Main()

  DIM Http AS HttpClient
  DIM sCad AS String

  Http = NEW HttpClient

  Http.Async = FALSE
  Http.TimeOut = 10
  Http.URL = "http://gambas.gnulinex.org/gtk/"

  Http.Get()

  IF Http.Status < 0 THEN
    PRINT "Error al recibir la página"
  ELSE
    READ #Http, sCad, Lof(Http)
    PRINT sCad
  END IF

  CLOSE #Http

END
```

En primer lugar definimos y creamos un objeto de la clase HttpClient llamado Http. Situamos su propiedad Async a FALSE para que el proceso sea síncrono, es decir, que el cliente HTTP quede bloqueado mientras se recibe la página. Puesto que el bloqueo podría durar un tiempo excesivo, definimos también con la propiedad TimeOut un tiempo máximo en segundos antes de dar por fracasada la conexión y recepción de datos.

Indicamos la URL que contiene el nombre del servidor y el documento dentro de éste.

Llamamos al método Get con el fin de recibir la página, con lo que el progran interrumpido hasta su recepción hasta un error o hasta que pasen 10 segund máximo.

Leemos el valor de la propiedad Status que, como en el caso de los sockets o te DNS, tendrá valor mayor que cero cuando está activo, cero en caso de éxito que cero en caso de error. Si hay un error, lo indicamos por la consola. Si bien, leemos, como en el caso de los sockets o cualquier otro flujo, empleand trucción READ y mostramos el contenido de la página web recibida por co

Finalmente, cerramos el cliente Http. Esto es necesario, ya que el cliente mantener viva la conexión con el servidor mientras le es posible, para acc esta manera la recepción de múltiples documentos de un mismo servidor.

> El protocolo HTTP establece un tiempo en el que el servidor mantiene el socket to con el cliente. Si necesitamos recibir varias páginas de un servidor o realizar peticiones POST consecutivas, deberíamos emplear la instrucción CLOSE sólo a de todo el proceso, con lo cual se ganará en velocidad y se utilizarán menos sos del sistema.

En cuanto al método asíncrono, el modo de proceder es exactamente el mis con los sockets: esperar al evento *Read* para ir leyendo fragmentos del doc en proceso de recepción, o al evento *Error* para atender posibles problemas de nicación con el servidor.

Al margen de los problemas físicos de comunicación, que se detectan con el *Error*, el propio protocolo HTTP puede especificar códigos de error en los que la nicación/recepción de datos ha sido perfecta a nivel físico, pero se han pro

gicos. El caso más común es solicitar un documento que no existe en el en cuyo caso se genera el error *404*. Otros errores comunes pueden ser el so prohibido) o el *500* (error interno del servidor). Para detectar y tratar blemas, el cliente HTTP proporciona dos propiedades que se pueden con- *de*, que es numérica y devuelve el código de error (el valor *200* significa fue bien), y *Reason*, que es una cadena de texto explicativa del error, pro- da por el servidor.

devuelto en *Reason* depende del programa utilizado como servidor web y no gatorio. Por ello, a efectos de control de errores, debe emplearse sólo el valor co *Code* y usarse *Reason* sólo a efectos de información del usuario.

mentos HTTP se reciben empaquetados, de forma que existe un encabeza- ntiene *meta información* del servidor y un cuerpo donde se encuentra alo- ocumento en sí. La lectura mediante **READ** u otros métodos para flujos sólo ceso a los datos del cuerpo. Para obtener las cabeceras, que pueden ser úti- omprobar datos tales como la fecha y hora del servidor, el tipo de servidor nformaciones específicas, podemos recurrir a la propiedad **Headers**. Cada e *meta información* es una línea, por lo que **Headers** devuelve una matriz de cada una de las cuales es una información procedente del servidor. En este derivado del anterior se muestra al final los encabezados del servidor.

```
PUBLIC SUB Main()

  DIM Http AS HttpClient
  DIM sCad AS String

  Http = NEW HttpClient

  Http.Async = FALSE
  Http.TimeOut = 10
```

```
Http.URL = "http://gambas.gnulinex.org/gtk/"

Http.Get()

IF Http.Status < 0 THEN
  PRINT "Error al recibir la página"
ELSE
  READ #Http, sCad, Lof(Http)
  PRINT sCad

  PRINT "\n------------Metadatos--------\n"
  FOR EACH sCad IN Http.Headers
    PRINT sCad
  NEXT

END IF

CLOSE #Http

END
```


El protocolo HTTP permite al cliente autenticarse para recibir determinadas páginas no accesibles al público en general. El usuario ha de disponer de un nombre de usuario y contraseña. Para introducir estos valores, se ha de especificar el nombre de usuario en la propiedad *User* y la contraseña en la propiedad *Password*, antes de llamar a los métodos *Get* o *Post*.

Por otra parte existen diversos métodos de autenticación, desde los menos seguros (envío de usuario/clave sin codificar) hasta algunos algoritmos más seguros. Además del usuario y clave, se ha de especificar el método de autenticación rellenando el valor de la propiedad *Auth* con alguna de las constantes que representan los métodos soportados:

Net.AuthBasic
Net.AuthDigest
Net.AuthGSSNegotiate
Net.AuthNTLM

El desarrollador de la aplicación ha de conocer de antemano el método que necesita para acceder al servidor.

El protocolo HTTP también contempla el uso de cookies, que es información que el servidor guarda en el cliente y que es consultada de nuevo por el servidor en ocasiones posteriores, antes de enviar un documento al cliente. Puede servir, por ejemplo, para saber si la página ya ha sido visitada con anterioridad por dicho cliente. Por defecto, el cliente HTTP no acepta las cookies provenientes del servidor. Si se especifica una ruta a un archivo con la propiedad *CookiesFile*, éstas se activarán y el cliente HTTP se nutrirá de las cookies existentes en dicho archivo para devolver información al servidor. Si la propiedad *UpdateCookies* se sitúa a TRUE, se permitirá el acceso de escritura al fichero de cookies, con lo cual las nuevas cookies se guardarán entre ejecución y ejecución del programa.

Este componente se encuentra en desarrollo y futuras versiones contemplarán el uso de SSL y certificados.

6. 8 Protocolo FTP

Al igual que el protocolo HTTP, FTP se encuentra muy extendido a lo largo de Internet y está diseñado específicamente para la transmisión y recepción de ficheros. La estructura lógica de un servidor FTP es muy próxima a la de un sistema de archivos locales: existen una serie de carpetas con estructura arbórea y dentro de cada carpeta existen archivos.

El soporte de FTP en Gambas es actualmente muy sencillo y también bastante limitado. Permite, esencialmente, *subir* y *bajar* archivos del servidor. Con una sintaxis

muy similar a la del cliente HTTP, tendremos que indicar la URL deseada y llamar a los métodos *Get()* o *Put()*.

Al igual que en el cliente HTTP, se permite la gestión de nombre de usuario y acceso al servidor.

Futuras versiones de este cliente darán acceso a listados de carpetas y comandos personalizados. No obstante, como se ha indicado, ahora es una clase muy limitada y puede plantearse como alternativa el uso directo de utilidades como *ftp*, *curl* o *wget* para aplicaciones complejas que necesiten acceso a servidores FTP, dado que la gestión de procesos externos es muy sencilla con Gambas.

7. XML

● ○ Hoy en día es común oír hablar de XML como la solución a todos los problemas de gestión informática en la empresa. Es cierto, desde luego, que se encuentra sobrevalorado, posiblemente debido a las campañas de marketing realizadas por grandes compañías de software.

No obstante, también es cierto que XML presenta algunas ventajas importantes a la hora de intercambiar datos entre diferentes sistemas. Pero ¿qué es XML? Pues en realidad no es nada nuevo ni revolucionario, se trata simple y llanamente de una definición de formato para cualquier tipo de documentos.

Cuando necesitamos escribir un fichero de configuración, un documento con registros extraídos de una base de datos, comunicar datos a un dispositivo, un formato para un fichero de texto o una hoja de cálculo, siempre se le plantea el mismo

problema al programador: la necesidad de un formato, documentado en el caso ideal, que defina el modo en que esa información se inserta en un fichero o en un flujo de datos a través de una red, con el fin de que los programas origen y destino de esos datos puedan escribir o leer cada parte del documento correctamente. Muchas veces escribir un fichero con un determinado formato es una tarea casi trivial, pero el proceso inverso, la lectura, suele ser engorroso, dado que hay que comprobar errores, trocear cadenas, comprobar la validez de cada fragmento, etc.

Por otra parte, cada aplicación ha desarrollado hasta hace poco sus propios formatos para el almacenamiento y lectura de la información que maneja, haciendo muy difícil eliminar un elemento de una cadena de programas para reemplazarlo por una aplicación nueva. Esto tiene una importancia grande cuando una empresa se plantea el paso de sistemas propietarios a los que se encuentran encadenados, hacia otros libres donde obtienen ventajas de precio, y capacidad de elección de proveedores y servicios más justos por la inversión realizada. Los formatos propietarios atan literalmente al cliente a los servicios y deseos de una empresa proveedora.

XML proporciona ayuda en todos los aspectos comentados: es un estándar que define cómo se han de insertar los datos y los campos en un fichero o flujo. Además, las herramientas de gestión de documentos XML proporcionan las funciones necesarias para leer, escribir y verificar los datos embebidos en dichos documentos. Por último, se trata de un estándar accesible a todos los programadores y casas de software, lo que proporciona libertad para manejar y comprender el contenido de los datos.

El formato de un documento XML es similar a uno escrito con HTML, no obstante hay diferencias sustanciales. La primera, y más importante, es que XML es un formato de carácter general, pensado para trabajar con cualquier tipo de datos, mientras que HTML se encuentra limitado al diseño de páginas web. La segunda es que, a diferencia de HTML, donde hay etiquetas, como la de párrafo nuevo (*<p>*) que se suelen dejar abiertas, en XML absolutamente todas las etiquetas deben estar anidadas, y cada etiqueta abierta debe cerrarse. Por otra parte, XML es sensible a mayúsculas, es decir, una sección que comienza con la etiqueta *<p>* no es igual a otra que comienza con la etiqueta *<P>*.

El aspecto de un fichero XML simple, puede ser éste:

```
<?xml version="1.0"?>
<datos>
 <usuario>
  <nombre>Eric Smith</nombre>
  <socio>113</socio>
 </usuario>
</datos>
```

Los documentos XML siempre comienzan, en la actualidad, por la cadena **<?xml version="1.0"?>**, en la cual se especifica que a continuación vienen datos con formato XML. Aunque esta etiqueta puede ser obviada, siempre es conveniente añadirla, ya que contiene información importante. En este caso se especifica la versión de XML que se está utilizando, la **1.0**, y que es la única que se emplea en la actualidad. Si en el futuro una nueva especificación internacional de XML ampliara o modificara XML, con una versión 2.0, por ejemplo, disponer de esa etiqueta en un documento almacenado hace meses o años garantizará que los programas sigan sabiendo cómo interpretar lo que contiene un documento. Por otra parte, los documentos XML, por defecto, emplean la codificación de caracteres UTF-8. Si por cualquier razón hemos de tratar con documentos XML que empleen otra codificación, también encontraremos esa información en la etiqueta inicial:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

A continuación, llega el cuerpo del documento XML. Hay una etiqueta inicial que da nombre al documento, en este caso simplemente **datos**. Después, en este documento vienen los datos de los usuarios de una asociación: el nombre y el número de asociado. Todo ello encerrado entre etiquetas abiertas y cerradas, de modo que el documento queda ordenado en secciones, con varios niveles de anidación.

XML permite que las etiquetas dispongan de *atributos*. Por ejemplo, podemos prever que nuestro fichero de datos contemple un número de versión para futuras

ampliaciones, y que en cada usuario se añada el año en que entró a formar parte de la asociación. Los atributos tienen la forma:

```
definicion="valor"
```

Es decir, se indicará el nombre del atributo, un signo de igual y, encerrado entre comillas, el valor de ese atributo. El fichero podría ser ahora así:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<datos version="2.4">
 <usuario>
  <nombre inicio="2005">Eric Smith</nombre>
  <socio>113</socio>
 </usuario>
</datos>
```


Una etiqueta que se abra y cierre sin información en su interior, por ejemplo *<p></p>* para designar un párrafo en un documento XHTML, puede simplificarse escribiendo la etiqueta de apertura con la barra al final del nombre de la etiqueta: *<p/>*.

Estos son los elementos más básicos de XML, pero más allá existe una amplia gama de conceptos que quedan fuera del alcance de este manual, y que aportan un gran número de posibilidades a la gestión de documentos XML: existen etiquetas especiales para comentarios, un lenguaje para validación de datos, DTD, hojas de estilo con formato XSL, espacios de nombres, etc. Un buen punto de inicio para aprender XML es la web **http://www.w3schools.com/xml/default.asp**, donde se encuentran tutoriales al respecto, así como enlaces a otras fuentes de información.

7.1 Escritura con XmlWriter

La clase *XmlWriter*, contenida dentro del componente *gb.xml*, aporta un modo sencillo para escribir un documento XML. Supongamos que deseamos almacenar en

un programa los datos relativos a varias conexiones telefónicas a Internet. Nos interesará el nombre de la conexión, si está disponible en todo el territorio nacional o está limitada a una localidad, el teléfono y las DNS, primaria y secundaria, aportadas por el proveedor.

Plantearemos un documento en el cual tendremos una sección *conexion* para cada conexión, que tendrá un atributo *nombre* y otro *local*, este último con dos valores: 1, si es local, o 0, si es nacional. Dentro de la sección *conexion* habrá un campo para almacenar el número de teléfono, así como otro para los DNS primario y secundario, los cuales tendrán un atributo que determina si es el *primario*.

> Siempre es conveniente emplear nombres de etiquetas y atributos que no contengan acentos o caracteres especiales de cada idioma (como ñ o ç), para evitar problemas si otros programadores que deban retocar el programa o el fichero tienen teclados distintos o emplean *parsers* de XML pobres que ignoren algunos aspectos de la codificación.

Éste será el aspecto del fichero XML del ejemplo:

```
<?xml version="1.0"?>
<conexiones version="1.0">
 <conexion id="castuonet" local="0">
  <telefono>1199212321</telefono>
  <dns primario="1">127.0.0.2</dns>
  <dns primario="0">127.0.0.3</dns>
 </conexion>
 <conexion id="limbonet" local="0">
  <telefono>229943484</telefono>
  <dns primario="1">127.0.10.10</dns>
  <dns primario="0">127.0.20.42</dns>
 </conexion>
</conexiones>
```

Antes de continuar, observemos el formato. Está separado en varias líneas, o retornos de carro, y existe una indentación para indicar los distintos niveles de anidación. Esto no es en absoluto necesario, y los *parsers* de XML ignoran los espacios en blanco y tabulaciones, así como los retornos de carro. Esto es sólo una ayuda para quien tenga que consultar o modificar el fichero desde un editor de texto común. Sólo tiene, por tanto, sentido a nivel humano, y un fichero XML correcto podría estar contenido en una sóla línea de texto. Si estamos acostumbrados a programar en C o C++, sabremos que con este lenguaje ocurre lo mismo: las diferentes líneas y tabulaciones dan orden y legibilidad al programa, pero el mismo podría escribirse en una sola línea. En C, las llaves y puntos y comas son significativas, no así los espacios o retornos de carro. En XML lo significativo son las aperturas y cierres de etiquetas.

A continuación vamos a crear un programa de consola llamado **EscribeXML**, y dentro de él un único módulo **modMain** con una función *Main*.

El programa tendrá una referencia al componente **gb.xml**.

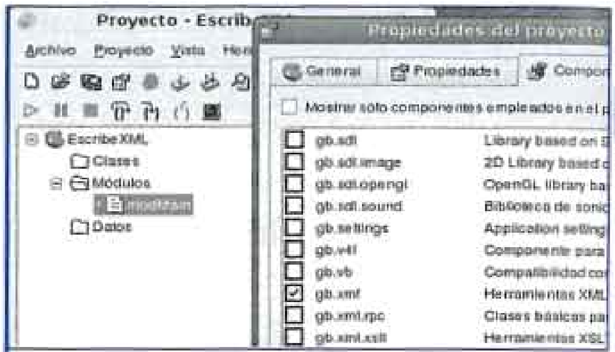


Figura 1. Proyecto **EscribeXML**.

Definimos y creamos un objeto **XmlWriter**. Tras esto, lo abrimos con **Open** para comenzar la escritura. Este método acepta tres parámetros: el primero, obligatorio, es el nombre del fichero a escribir, que puede ser una ruta dentro del sistema de archivos o una cadena en blanco, en cuyo caso se escribirá en memoria, y luego podremos obtenerlo como una cadena para, por ejemplo, enviarlo por la red a un equipo remoto. El segundo es opcional, e indica que deseamos que tenga indentación (como se comentó antes, para aumentar su legibilidad), o bien si se escribirá en una sola línea, para ahorrar espacio, a cambio de hacerlo ilegible desde un editor no especializado. El tercero, también opcional, sirve para, si se desea, especificar la codificación, diferente de UTF-8.

```
' Gambas module file

PUBLIC SUB Main()

  DIM Xml AS XmlWriter

  Xml = NEW XmlWriter
  Xml.Open("", TRUE)
```

Indicar una codificación diferente sirve para que ésta quede reflejada al inicio del documento, pero no realizará por nosotros la conversión de las cadenas que escribamos, tarea que habremos de hacer con funciones como *Conv$*, si procede.

Escribimos la primera sección. Hemos de introducir una etiqueta de apertura, tarea que se realiza con el método StartElement:

```
Xml.StartElement("conexiones")
```

Además dispone de un atributo "version" con valor "1.0":

```
Xml.Attribute("version","1.0")
```

En este caso también podemos unir las dos instrucciones anteriores en una sola: StartElement acepta un parámetro opcional, consistente en una matriz de cadenas que sean pares de valores atributo-valor del atributo:

```
Xml.StartElement("conexiones", ["version", "1.0"])
```

Dentro de esta sección tenemos cada una de las conexiones definidas. Comenzamos por la primera que, al igual que antes, es una etiqueta de apertura, con un nombre y dos atributos, en este caso:

```
Xml.StartElement("conexion", ["id", "castuonet",
"local", "0"])
```

Dentro de esta región, disponemos de una nueva apertura de etiqueta para el teléfono:

```
Xml.StartElement("telefono")
```

Contiene un texto que comprende el número de teléfono:

```
Xml.Text("1199212321")
```

Tras este paso cerramos la etiqueta:

```
Xml.EndElement()
```


No obstante, podemos resumir las tres instrucciones anteriores en una sola, válida para estos elementos simples que tan sólo contienen un texto dentro (o nada) y luego se cierran.

```
Xml.Element("telefono", "1199212321")
```

Para los DNS iniciamos una etiqueta "dns" con un atributo:

```
Xml.StartElement("dns", ["primario", "1"])
```

Incluimos el texto con la IP:

```
Xml.Text("127.0.0.2")
```

Y finalizamos la sección:

```
Xml.EndElement()
```

Lo mismo para el segundo DNS:

```
Xml.StartElement("dns", ["primario", "0"])
Xml.Text("127.0.0.3")
Xml.EndElement()
```

Finalmente, cerramos la etiqueta "**conexion**" que contiene los elementos anteriores relativos a esa conexión:

```
Xml.EndElement()
```

Procedemos de igual manera para la segunda conexión, tal y como podemos observar en el siguiente código:

```
Xml.StartElement("conexion", ["id", "limbonet", "local",
"1"])
Xml.Element("telefono", "229943484")
Xml.StartElement("dns", ["primario", "1"])
Xml.Text("127.0.10.10")
Xml.EndElement()
Xml.StartElement("dns", ["primario", "0"])
Xml.Text("127.0.20.42")
Xml.EndElement()
Xml.EndElement()
```

Finalmente cerramos el documento para que se escriba. Si hubiésemos especificado un nombre de fichero, sería ahora cuando se volcaría al fichero al haberlo hecho, en nuestro caso, en memoria.

La llamada a **EndDocument** nos devuelve una cadena con el documento XML, que mostramos por la consola:

```
PRINT Xml.EndDocument()
```

El programa completo queda de la siguiente manera:

```
' Gambas module file

PUBLIC SUB Main()

  DIM Xml AS XmlWriter

  Xml = NEW XmlWriter

  Xml.Open("", TRUE)

  Xml.StartElement("conexiones", ["version", "1.0"])

  Xml.StartElement("conexion", ["id", "castuonet",
  "local", "0"])
  Xml.Element("telefono", "1199212321")
  Xml.StartElement("dns", ["primario", "1"])
  Xml.Text("127.0.0.2")
  Xml.EndElement()
  Xml.StartElement("dns", ["primario", "0"])
  Xml.Text("127.0.0.3")
  Xml.EndElement()
  Xml.EndElement()

  Xml.StartElement("conexion", ["id", "limbonet",
  "local", "1"])
  Xml.Element("telefono", "229943484")
  Xml.StartElement("dns", ["primario", "1"])
  Xml.Text("127.0.10.10")
  Xml.EndElement()
  Xml.StartElement("dns", ["primario", "0"])
  Xml.Text("127.0.20.42")
```

```
    Xml.EndElement()
    Xml.EndElement()

    PRINT Xml.EndDocument()

END
```

l ejecutarlo, veremos el documento por la consola. Lo modificamos para que se scriba en un fichero en lugar de en la consola, lo cual nos servirá para leerlo a connuación en un ejemplo de lectura de fichero XML. Para ello modificaremos el méto-o **Open**:

```
Xml.Open(User.Home & "/conexiones.xml", TRUE)
```

' eliminaremos la instrucción *PRINT* al final:

```
Xml.EndDocument()
```

n este caso, la llamada al método **EndDocument** volcará el documento en un fiche-ɔ dentro de nuestra carpeta personal, llamado **conexiones.xml**.

> El fichero abierto con *Open* no se vuelca realmente al disco duro hasta la llamada a *EndDocument*. A su vez, el método *EndDocument* se encarga de cerrar todas las etiquetas que hubieran quedado abiertas, para garantizar la coherencia XML de éste.

exploramos la clase *XmlWriter*, observaremos que, además de los métodos mentados, aporta otros para escribir comentarios y elementos correspondien-al lenguaje DTD, entre otras características. También dispone de los métodos áticos *Base64* y *BinHex*, que convierten una cadena a las codificaciones con estos mbres. Dichas codificaciones permiten introducir datos binarios dentro de un ıero XML.

7. 2 Lectura con XmlReader

Modelos de lectura

Al menos existen tres métodos para leer los contenidos de un fichero XML:

1. Un método se basa en leer el fichero de principio a fin. El lector va generando eventos conforme se entra y sale de los distintos nodos del documento, y los gestores de eventos escritos por el programador van recibiendo la información. Esta forma de trabajo aún no se ha implementado en el componente *gb.xml*, aunque se prevé su inclusión en futuras versiones.

2. Otro método consiste en cargar el documento completo en memoria, para luego navegar por él, con lo cual se obtiene gran flexibilidad a costa de un consumo considerable de recursos del sistema. Este método está parcialmente implementado en Gambas a través de la clase *XmlDocument*, si bien su finalización no está prevista hasta futuras versiones, y no se recomienda su empleo para lectura de ficheros XML. No obstante, esta clase ya se emplea para transformaciones XSLT, como veremos más adelante en este capítulo.

3. Por último, el método que consume menos recursos y aporta bastante simplicidad de aprendizaje y uso, es disponer de un cursor, que sólo se mueve hacia adelante, de nodo en nodo, y que en cada momento podemos emplear para conocer el contenido y tipo de cada nodo. Si hemos trabajado con la plataforma .NET(TM) o Mono(TM), nos será familiar la clase *XmlReader* y sus derivadas, como *XmlTextReader*, que trabajan de la misma manera. Este modo de trabajo se encuentra perfectamente soportado en el componente *gb.xml* a través de la clase *XmlReader*.

Planteamiento inicial

En el caso de Gambas, la lectura de documentos la realizaremos utilizando objetos de la clase *XmlReader*.

El código de lectura resultará más complejo, al tener en cuenta varios aspectos:

- Rechazar cada fichero no válido: puede haber ficheros con formato no XML o que contienen datos sin sentido para nuestra aplicación.

- Ignorar datos no conocidos: es posible que un documento contenga datos que no nos interesan, pero se han añadido al fichero por otra aplicación en previsión de futuros usos (puede haber, por ejemplo, una etiqueta <*tarifa*> dentro de cada conexión). También una etiqueta conocida puede contener atributos desconocidos.

- Orden desconocido: en un fichero XML no es relevante el orden en que se escriben los nodos hijos de un nodo, es decir, que estos dos ejemplos deberían ser dados por válidos:

```
<conexion>
     <dns>...
    <telefono>...
    <dns>..
</conexion>

<conexion>
    <telefono>....
    <dns>...
    <dns>...
</conexion>
```

Si la aplicación espera encontrar el nodo **telefono** antes del nodo **dns**, fallará al tratar el primer fichero, que, sin embargo, contiene la misma información.

- Ignorar etiquetas sin interés para nuestra aplicación: XML, como indicamos brevemente al principio, prevé la posibilidad de añadir comentarios (similares a los comentarios de cualquier programa, sin uso para éste pero que aumentan la legibilidad), nodos DTD, etc. Habremos de pasar sobre estos nodos ignorándolos y sin presuponer si existen o no.

Cuantas más posibilidades añadamos a nuestro código de salvar lo desconocido, más flexible haremos nuestro lector XML para permitir la lectura de datos provenientes, tal vez, de programas escritos por varios programadores con los que no tenemos contacto, o que tienen pensamientos algo diferentes acerca del contenido del fichero.

Un ejemplo de lectura

Crearemos un proyecto gráfico llamado **LeeXML**, con un formulario **FMain**, que tendrá la propiedad *Arrangement* con el valor *Fill*, y un único control *TreeView* en su interior llamado **Arbol**. El programa contendrá una referencia al componente **gb.xml**.

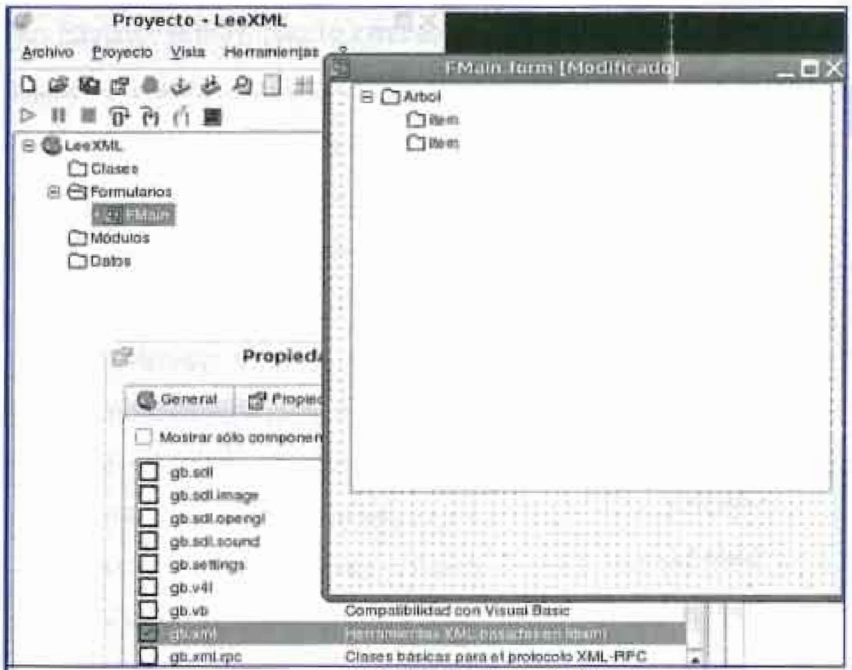


Figura 2. Proyecto **LeeXML**.

En la apertura del formulario leeremos el fichero XML. El método **Form_Open** quedará así:

- En primer lugar definimos el objeto **XmlReader**, lo creamos y tratamos de abrir el fichero XML. Si el fichero no existe, o no atiende a este formato, se generará un error en ese punto.

```
PUBLIC SUB Form_Open()

  DIM Xml AS XmlReader
  Xml = NEW XmlReader
  TRY Xml.Open(User.Home & "/conexiones.xml")
  IF ERROR THEN
    Message.Error("Fallo al abrir el fichero indicado")
    RETURN
  END IF
```

- Entramos en un bucle en el que leemos cada nodo avanzando por el contenido del fichero. Nos interesa encontrar el primero de tipo **Element** y que su nombre sea **conexiones**. De no ser así, el fichero no contendría datos de interés y lo rechazaríamos. Pero si es correcto, llamaremos a una función **RellenaArbol**, donde trataremos estos datos.

```
  DO WHILE TRUE

    IF Xml.Node.Type = XmlReaderNodetype.Element THEN

      IF Xml.Node.Name = "conexiones" THEN

        RellenaArbol(Xml)

      ELSE

        Message.Error("El documento no contiene datos
        de conexiones")
        Xml.Close()

      END IF

    END IF
```

Por cada iteración del bucle, empleamos el método **Read**, que sitúa el puntero interno en el siguiente nodo del fichero XML. En este proceso, puede darse un error si el puntero llega a una zona donde el fichero está corrupto, es decir, que no cumple la norma XML y, por tanto, no puede ser leído.

```
TRY Xml.Read()
IF ERROR THEN
  Message.Error("Formato XML no válido")
  RETURN
END IF
```

Si llegamos al final del fichero (tras el último nodo), terminamos el bucle. Esta circunstancia se puede conocer porque la propiedad **Eof** del objeto *XmlReader* toma el valor *TRUE*.

```
  IF Xml.Eof THEN BREAK

LOOP
```


Tras la lectura del fichero, cerramos el objeto *XmlReader*.

```
  TRY Xml.Close()


END
```

Vamos ahora a implementar el procedimiento **RellenaArbol**. Entramos de nuevo en el bucle y, en primer lugar, leemos el siguiente nodo para situarnos dentro de *conexiones*.

En este caso habremos de seguir leyendo por cada uno de los elementos *conexion* que existen dentro de la etiqueta principal *conexiones*, y salir de la función cuando encontremos el final de esta etiqueta:

```
PUBLIC SUB RellenaArbol(Xml AS XmlReader)

 DO WHILE TRUE

  TRY Xml.Read()
  IF ERROR THEN RETURN
```

Si encontramos un nodo de tipo **Element** que se llame **conexion**, llamaremos a una función llamada **RellenaItem** para tratarlo. Pero si su nombre es desconocido para nosotros, lo ignoraremos, saltando todo su contenido para llegar al siguiente nodo del mismo nivel, con el método Next.

```
  IF Xml.Node.Type = XmlReaderNodeType.Element THEN

   IF Xml.Node.Name = "conexion" THEN

    RellenaItem(Xml)

   ELSE

    TRY Xml.Next()
    IF ERROR THEN BREAK

   END IF

  ELSE

   IF Xml.Node.Name = "conexiones"
      IF Xml.Node.Type = XmlReaderNodeType.
      EndElement THEN BREAK
   END IF

  END IF
```

```
    LOOP

END
```

RellenaItem es el procedimiento más complejo, en el cual leeremos el contenido de cada conexión exitente.

```
PUBLIC SUB RellenaItem(Xml AS XmlReader)

  DIM Limite AS Integer
  DIM sNodo AS String
  DIM sLocal AS String
  DIM sPrim AS String
```

En primer lugar vamos a recoger los datos de los atributos de la etiqueta *conexion*. Para ello, hemos de iterar con la propiedad **Attributes** del nodo. A lo largo del proceso de iteración iniciado con FOR EACH, el puntero interno del lector XML pasará por cada uno de los atributos del nodo, que a su vez son nodos, cuyo nombre y valor es el nombre y valor del atributo. Finalizado el proceso de iteración, el puntero vuelve al nodo sobre el que estábamos situados, y con los datos recabados, añadimos en nuestro control *TreeView*, un nodo al efecto para su representación gráfica.

```
  FOR EACH Xml.Node.Attributes

    IF Xml.Node.Name = "id" THEN sNodo = Xml.Node.Value
    IF Xml.Node.Name = "local" THEN sLocal = Xml.Node.
    Value

  NEXT

  IF sNodo <> "" AND sLocal <> "" THEN

    IF sNodo = "0" THEN
```

```
      TRY Arbol.Add(sNodo, sNodo & " (local)")
    ELSE
      TRY Arbol.Add(sNodo, sNodo & " (nacional)")
    END IF

  END IF
```

Pasamos ahora al interior del nodo *conexion* para extraer información de sus nodos *hijo*, es decir, las DNS y el número de teléfono. Buscaremos nodos de tipo **Element** y en función de su nombre actuaremos de un modo u otro.

```
TRY Xml.Read()
  IF ERROR THEN RETURN

  DO WHILE TRUE

    IF Xml.Node.Type = XmlReaderNodeType.Element THEN

      SELECT CASE Xml.Node.Name
```

Para el caso del teléfono, pasaremos del nodo actual (la etiqueta **teléfono**), al siguiente nodo que contendrá el texto con el número de teléfono, (podríamos, no obstante, mejorar el algoritmo contemplando la posibilidad de encontrar algo distinto a un nodo de texto, cosa que no haremos aquí por no complicar más el código, que siempre puede resultar complejo al principio).

```
        CASE "telefono"
          TRY Xml.Read()
          TRY Arbol.Add(sNodo & "-tel", "tel: " &
          Xml.Node.Value, NULL, sNodo)
```

Si el nodo es de tipo "**dns**", tendremos que comprobar el valor del atributo que indica, si es DNS primario o no, y luego leer el texto que contiene la IP:

```
CASE "dns"
   sPrim = "0"
   FOR EACH Xml.Node.Attributes
     IF Xml.Node.Name = "primario" THEN sPrim =
     Xml.Node.Value
   NEXT
   TRY Xml.Read()
   IF sPrim = "0" THEN
     TRY Arbol.Add(sNodo & "-dns2", "-dns2" &
     Xml.Node.Value, NULL, sNodo)
   ELSE
     TRY Arbol.Add(sNodo & "-dns1", "-dns1" &
     Xml.Node.Value, NULL, sNodo)
   END IF

 END SELECT
```


Una vez leído el nodo, pasamos al siguiente y continuamos leyendo hasta encontrar uno de tipo **EndElement**, donde sabremos que hemos encontrado el final del nodo *conexion*.

```
TRY Xml.Next()
IF ERROR THEN BREAK

 LOOP

ELSE

 IF Xml.Node.Type = XmlReaderNodeType.EndElement
 THEN BREAK


END IF
```

```
    TRY Xml.Read()
   IF ERROR THEN BREAK
  LOOP

END
```

e es el código completo expuesto:

```
' Gambas class file



PUBLIC SUB RellenaItem(Xml AS XmlReader)

  DIM Limite AS Integer
  DIM sNodo AS String
  DIM sLocal AS String
  DIM sPrim AS String

  FOR EACH Xml.Node.Attributes

    IF Xml.Node.Name = "id" THEN sNodo = Xml.Node.Value
    IF Xml.Node.Name = "local" THEN sLocal = Xml.Node.
    Value

  NEXT

  IF sNodo <> "" AND sLocal <> "" THEN

    IF sNodo = "0" THEN
      TRY Arbol.Add(sNodo, sNodo & " (local)")
    ELSE
      TRY Arbol.Add(sNodo, sNodo & " (nacional)")
    END IF
```

```
END IF

TRY Xml.Read()
IF ERROR THEN RETURN

DO WHILE TRUE

  IF Xml.Node.Type = XmlReaderNodeType.Element THEN

    SELECT CASE Xml.Node.Name

      CASE "telefono"
        TRY Xml.Read()
        TRY Arbol.Add(sNodo & "-tel", "tel: " & Xml.
        Node.Value, NULL, sNodo)

      CASE "dns"
          sPrim = "0"
          FOR EACH Xml.Node.Attributes
            IF Xml.Node.Name = "primario" THEN sPrim
            = Xml.Node.Value
          NEXT
          TRY Xml.Read()
          IF sPrim = "0" THEN
            TRY Arbol.Add(sNodo & "-dns2", "dns2: " &
            Xml.Node.Value, NULL, sNodo)
          ELSE
            TRY Arbol.Add(sNodo & "-dns1", "dns1: " &
            Xml.Node.Value, NULL, sNodo)
          END IF


    END SELECT
```

```
      TRY Xml.Next()
      IF ERROR THEN BREAK

    ELSE

      IF Xml.Node.Type = XmlReaderNodeType.EndElement
      THEN BREAK

    END IF

    TRY Xml.Read()
    IF ERROR THEN BREAK

  LOOP
END

PUBLIC SUB RellenaArbol(Xml AS XmlReader)

  DO WHILE TRUE

    TRY Xml.Read()
    IF ERROR THEN RETURN

    IF Xml.Node.Type = XmlReaderNodeType.Element THEN

      IF Xml.Node.Name = "conexion" THEN

        RellenaItem(Xml)

      ELSE

        TRY Xml.Next()
        IF ERROR THEN BREAK
```

```
        END IF

      ELSE

        IF Xml.Node.Name = "conexiones" AND Xml.Node.Type
        = XmlReaderNodeType.EndElement THEN
          BREAK
        END IF

      END IF

    LOOP

  END

  PUBLIC SUB Form_Open()

    DIM Xml AS XmlReader

    Xml = NEW XmlReader

    TRY Xml.Open(User.Home & "/conexiones.xml")
    IF ERROR THEN
      Message.Error("Fallo al abrir el fichero indicado")
      RETURN
    END IF


    DO WHILE TRUE

      IF Xml.Node.Type = XmlReaderNodetype.Element THEN

        IF Xml.Node.Name = "conexiones" THEN
```

```
            RellenaArbol(Xml)

          ELSE

            Message.Error("El documento no contiene datos
            de conexiones")
            Xml.Close()

          END IF

        END IF

        TRY Xml.Read()
        IF ERROR THEN
          Message.Error("Formato XML no válido")
          RETURN
        END IF

        IF Xml.Eof THEN BREAK

      LOOP

      TRY Xml.Close()

    END
```

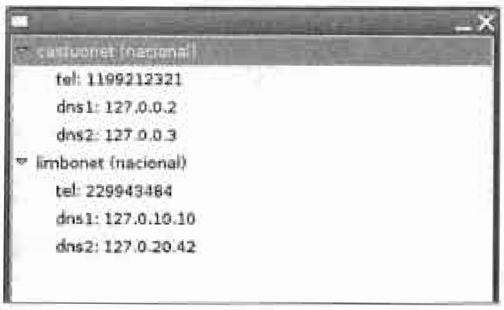



Figura 3. Resultado del fichero XML.

Este código dará lugar, al ejecutarse, a una visión en árbol de los datos contenidos en el fichero XML.

7. 3 XSLT

¿Qué es XSLT?

Acompañando a XML, XSLT permite realizar conversiones de formatos de documentos. Con XSLT se puede, por ejemplo, convertir datos de un documento XML en un documento HTML, o cosas más complejas como generar un documento PDF o StarWriter a partir de datos XML que nosotros hayamos diseñado.

Gracias a XSLT se puede separar de modo definitivo la información de su representación, por lo cual se emplea extensivamente en aplicaciones web, que pueden recibir datos de una base remota en formato XML y convertirlos, generalmente, a HTML para enviarlos al cliente con una representación agradable.

XSLT se basa en unos documentos, con formato XML, llamados plantillas, que contienen las instrucciones necesarias para convertir un determinado documento XML (con las etiquetas y atributos propios de dicho documento) en otro con distinto formato.

XSLT es extenso para tratarlo aquí en profundidad aunque, como siempre ocurre con los estándares abiertos y casi nunca con los formatos propietarios, podemos encontrar fuentes específicas de información adicional, por ejemplo en http://www.w3schools.com.

Una plantilla de ejemplo

Cada plantilla XSLT se refiere al contenido de un determinado documento XML. Supongamos un documento XML como éste, en el que se encuentra un listado de socios:

```
<?xml version="1.0"?>
<socios>
  <socio>
    <numero>1123</numero>
    <nombre>Juan L. Aguilar</nombre>
```

```
    <tipo>Honorario</tipo>
  </socio>
  <socio>
    <numero>2135</numero>
    <nombre>Salvador G. Tierra</nombre>
    <tipo>Regular</tipo>
  </socio>
  <socio>
    <numero>9654</numero>
    <nombre>Alberto N. Parra</nombre>
    <tipo>Regular</tipo>
  </socio>
</socios>
```

Las plantillas XSLT siempre comienzan con unos identificadores. El primero, de documento XML, ya lo conocemos, el segundo denota que lo que viene a continuación es un documento XSLT.

```
<?xml version="1.0">
<xsl:stylesheet version="1.0" xmlns:xsl="http://
www.w3.org/1999/XSL/Transform">
```

Tras esto, se escribe el código en sí, en el cual generaremos una tabla HTML con los datos.

```
<xsl:template match="/">
  <html>
  <body>
    <h1>Listado de socios</h1>
    <table border="1">
      <tr>
        <th><b>Nro.</b></th>
        <th><b>Nombre</b></th>
```

```
            <th><b>Tipo</b></th>
          </tr>
          <xsl:for-each select="socios/socio">
          <tr>
            <td><xsl:value-of select="numero"/></td>
            <td><xsl:value-of select="nombre"/></td>
            <td><xsl:value-of select="tipo"/></td>
          </tr>
          </xsl:for-each>
        </table>
      </body>
      </html>
    </xsl:template>
    </xsl:stylesheet>
```


Como podemos observar, vamos embebiendo las etiquetas HTML; empleamos el iterador **for each** para tomar cada elemento del fichero XML; y situamos en cada punto de la tabla uno de los campos elegidos.

Transformando el documento con Gambas

Hasta aquí, lo que tenemos es un documento XML y una plantilla XSLT, pero ahora necesitamos un motor que realice la conversión. Para ello guardaremos el fichero con datos en nuestra carpeta personal como **socios.xml**, y la plantilla como **socios.xsl**. Creamos un nuevo proyecto de consola llamado **TransformaXSLT**, con un único módulo *modMain* y una referencia al componente *gb.xml.xslt*.

El código será tan simple como éste:

```
' Gambas module file

PUBLIC SUB Main()

  DIM Documento AS NEW XmlDocument
```

```
        DIM Plantilla AS NEW XmlDocument
        DIM Resultado AS XmlDocument

        Documento.Open(User.Home & "/socios.xml")
        Plantilla.Open(User.Home & "/socios.xsl")

        Resultado = XSLT.Transform(Documento, Plantilla)

        Resultado.Write(User.Home & "/socios.html")

END
```

ɔmo indicamos anteriormente, la clase **XmlDocument** carga y verifica un documento VIL en memoria. En este caso cargamos dos documentos: el primero, llamado **ɔcumento**, contiene los datos de los socios; el segundo, **Plantilla**, es la hoja XSLT que dica cómo transformarla en HTML. La única clase que aporta el componente *gb.xml.xslt*, llamada **XSLT**, es estática y dispone de un único método **Transform**, al cual pasamos como parámetros el documento y la plantilla, y devuelve un documento nuevo con el formato indicado, en este caso una página web. Escribimos dicha página en un fichero en nuestra carpeta personal, y salimos. Si consultamos con el navegador la página obtenida, veremos un resultado como el que se muestra en la figura de la izquierda.

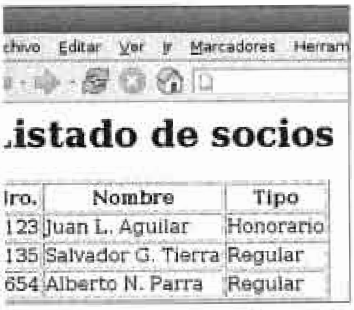


Figura 4. Resultado de la página obtenida.

7. 4 Acerca de XML-RPC

futura versión Gambas 2, dispondrá también de un componente XML-RPC. Estas las se refieren al uso de XML como sistema para comunicar dos procesos en dos iquinas diferentes. XML-RPC es un subconjunto de XML que define un lenguapara llamar a procesos remotos.

Un servidor XML-RPC acepta llamadas remotas a sus métodos. El proceso es muy similar a llamar a una función local dentro de un programa: el cliente llama a la función pasando unos parámetros (números enteros, cadenas, estructuras de datos...); y el servidor procesa la llamada y devuelve al cliente un resultado, que también será una cadena, número, fecha, etc.

El componente XML-RPC aportará varias facilidades para implementar estos procesos.

En el lado servidor dispondrá de un *parser*, en el cual definiremos los nombres de los métodos que se aceptarán, así como la correspondencia entre estos y otros locales del programa servidor. De este modo, tan sólo hay que pasar la petición del cliente al *parser*, el cual se encargará de verificar el número y tipo de los parámetros, y devolver un error al cliente, o llamar a la función local y devolver el resultado al cliente.

El servidor podrá funcionar en solitario, implementando un pequeño servicio web para atender las peticiones, o bien funcionar de modo controlado por la aplicación servidora, con el fin de implementar CGIs que se sirvan desde Apache, por ejemplo.

En la parte cliente es posible definir la URL del servidor y la forma de cada método. Igualmente podrá actuar en solitario, solicitando mediante una petición web la llamada al servidor. También se puede crear el documento XML de la petición, dejando a la aplicación el diseño del transporte y recepción de los datos con el servidor.

8. Herencia

8.1 Lenguajes orientados a objetos y herencia

Ya hemos visto una breve introducción al mundo de la programación orientada a objetos. Podemos decir que un objeto es una herramienta, por ejemplo, un túnel de lavado. La clase son los planos de esa herramienta, en los cuales están descritos los distintos cepillos giratorios, los conductos de agua y detergente, los secadores de aire caliente, los distintos sensores, el proceso automático paso a paso desde la entrada del coche hasta su salida, etc.

Cuando creamos un objeto, el sistema nos devuelve una herramienta, en este caso, un túnel de lavado dispuesto para funcionar, que se ha creado a partir de los planos que nosotros habíamos escrito en la clase.

Ahora bien, supongamos que como vendedores de túneles de lavado deseamos innovar y añadir una fase de encerado a nuestro túnel. Venderemos máquinas baratas sin encerado y otras más caras con esta funcionalidad añadida. La primera opción sería ponerse manos a la obra con nuevos planos desde cero, pero esto es costoso en tiempo y dinero. Lo mejor sería tomar los planos ya existentes y, a partir de ellos, añadir la nueva función.

En esto consiste precisamente la herencia: partimos de una clase ya escrita, probablemente grande y con mucha funcionalidad, y a partir de ella creamos otra clase (otros planos) que tiene el mismo rendimiento del original, más nuevas funciones que añadimos, la cual *hereda*, de ahí viene el término, la funcionalidad de su clase padre.

Además de añadir nuevas funcionalidades, podremos modificar también el comportamiento de las ya existentes para adaptarlas a la nueva *máquina a crear*, por ejemplo, los cepillos del túnel de lavado con cera tendrán que ser de un material que no se disuelva con los productos químicos que componen nuestra cera.

En resumen, vamos a explicar cómo crear clases que provienen de otras, que se comportan en primera instancia como la original y que añaden nuevas funcionalidades o modifican algunas de las existentes, eliminando así la necesidad de escribir las mismas líneas de código una y otra vez.

8. 2 Conceptos necesarios

La clase padre

Empecemos creando los planos de la máquina original, es decir, creemos una clase *padre*. Vamos a escribir una clase a partir de la cual crearemos objetos. Dichos objetos almacenan varios números reales y, una vez introducidos, nos devuelve la media aritmética de todos ellos.

Ahora, vamos a crear un nuevo proyecto de consola, lo llamamos **EstudioHerencia** y dentro de él un módulo de inicio llamado *modMain*, que nos servirá para las pruebas.

Además, creamos también una clase, llamada *ClsCalculo*, en la que escribiremos el código principal a estudiar.

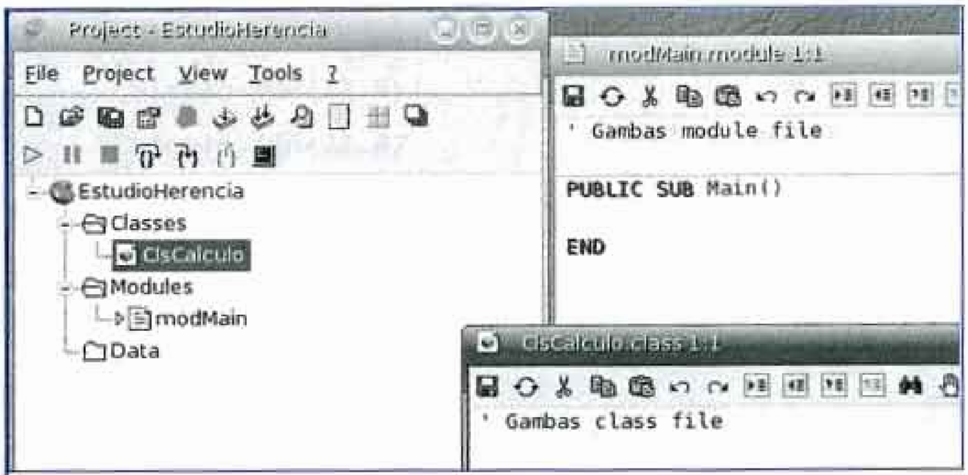


Figura 1. Proyecto nuevo de consola **EstudioHerencia**.

Dentro de *ClsCalculo* escribiremos el código necesario: tendremos una matriz privada de números reales, donde añadimos cada uno de los números a calcular y cuatro métodos: el primero, _New, es el constructor que sirve para inicializar la matriz; el segundo, _Free, libera la matriz al destruir el objeto; el tercero, **Add**, añade un nuevo valor a la serie; y el cuarto, **Average**, calcula la media aritmética de los números almacenados:

```
' Gambas class file
PUBLIC _Numbers AS Float[]

PUBLIC SUB _new()

  _Numbers = NEW Float[]

END
```

```
PUBLIC SUB _free()

  _Numbers.Clear()
  _Numbers=NULL

END

PUBLIC SUB Add(Vl AS Float)

  _Numbers.Add(Vl)

END

PUBLIC FUNCTION Average() AS Float

  DIM Nm AS Float
  DIM Vl AS Float

  IF _Numbers.Count = 0 THEN RETURN 0

  FOR EACH Vl IN _Numbers
    Nm += Vl
  NEXT

  RETURN Nm / _Numbers.Count

END
```

A la hora de realizar operaciones matemáticas, que la división por cero no está permitida, hemos de proveer un valor por defecto para ese caso o generar un error en tiempo de ejecución si procede.

Ahora escribimos algo de código adicional para probar nuestra clase, dentro de *modMain*. Nos limitamos a crear un objeto a partir de nuestra clase, introducir tres valores y mostrar en la consola la media de esos tres números:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT Calculo.Average()

END
```

Al ejecutarlo, obtendremos el valor 14.6 si todo fue bien.

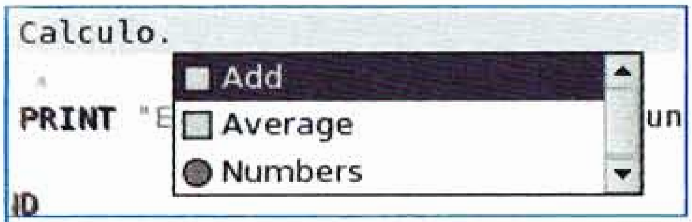


Figura 2. Ejemplo de matriz *Numbers*.

Observemos también que nuestra matriz tiene un nombre algo curioso **_Numbers**. La matriz ha de ser pública porque más tarde la emplearemos en la clase *hija*: si fuera privada, quedaría restringida al ámbito de la clase original y no podríamos usarla desde las clases *hijas*.

Ahora bien, si hubiésemos llamado a la matriz simplemente *Numbers*, el programador vería esta variable desde el sistema de autocompletado de código.

> Las clases *hijas* de otra clase no pueden emplear las funciones, propiedades o variables privadas de la clase padre.

Sin embargo, los métodos públicos que comienzan por el signo de subrayado no se muestran nunca en el sistema de autocompletado ni en el sistema de ayuda, cuando creamos componentes.



Figura 3. Variables públicas no visibles.

Puesto que en Gambas no existe el concepto de C++ y otros lenguajes de propiedad, variable o función *Protegida* (accesible sólo desde la clase y sus clases hijas), esta funcionalidad nos permite declarar variables públicas no visibles desde el código principal, manteniendo así una interfaz más coherente con nuestros propósitos.

Los nombres de variables, propiedades o funciones que comienzan con un subrayado, no se muestran ni en el sistema de autocompletado, ni en el de ayuda de Gambas.

NOTA PARA PROGRAMADORES DE C++

En Gambas existen propiedades, métodos y variables privadas o públicas, pero no protegidas.

La clase hija. Palabra clave INHERITS

Supongamos ahora que deseamos tener una nueva clase que se comporte como la inicial, pero tenga una propiedad adicional, de sólo lectura, que nos devuelve el número de elementos que hemos almacenado.

Crearemos, entonces, una nueva clase llamada *ClsCalculo2*, en la cual introduciremos al inicio la palabra clave *INHERITS* seguida del nombre de la clase *padre* (ver Figura 4).

Esto es todo lo necesario para tener una clase que hereda todas las propiedades de su clase *padre*. Probemos a modificar el código del método **Main()** de modo que creemos un objeto de la segunda clase, en lugar de la original.

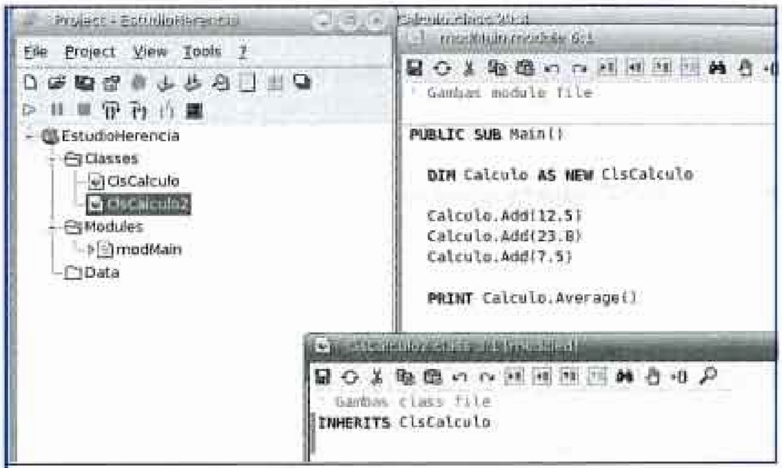


Figura 4. Nueva clase *ClsCalculo2*.

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo2


  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)


  PRINT Calculo.Average()
END
```

Si ejecutamos ahora el programa, observaremos que el resultado *14.6* es exactamente el mismo: la nueva clase ya dispone de todos los métodos, propiedades y eventos de la original, sin necesidad de escribir el código que los implemente.

Para escribir una clase que hereda las características de una clase *padre*, tecleamos **INHERITS** seguido del nombre de la clase *padre*, al inicio del código de la nueva clase:

```
' Gambas class file
INHERITS ClsCalculo
```

Nota para programadores de C++

En Gambas cada clase *hija* tiene una sola clase *padre*, no existe el concepto de herencia múltiple.

□□□□□ Extendiendo funcionalidades. Palabra clave SUPER

La nueva clase *ClsCalculo* tiene poco interés por ahora: hace lo mismo que la original. Añadamos entonces en la clase *hija* la nueva propiedad que nos devuelve el número de elementos almacenados:

```
' Gambas class file
INHERITS ClsCalculo

PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END
```

Estamos empleando una nueva palabra clave por primera vez, **SUPER**. Sabemos que cuando nos referimos a un objeto podemos utilizar, bien el nombre del objeto, o bien la palabra clave *ME* cuando nos referimos al objeto actual dentro de la propia clase. Pues bien, en este caso, la matriz **_Numbers** no se encuentra dentro de la clase *ClsCalculo2*, sino dentro de la clase *padre* **ClsCalculo.**

Con la palabra clave *SUPER* no nos referimos al objeto actual, sino al padre del objeto actual.

Escribamos ahora el código de la función **Main()** para aprovechar la nueva característica de nuestra clase *hija*:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo2

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()

END
```

Como es fácil adivinar, el resultado de la ejecución es:

```
Elementos: 3 - Media: 14.6.
```


□□□□□ Modificando funcionalidades

Hasta ahora tenemos dos clases, una de las cuales aporta una propiedad adicional *Count*, y sigue siendo poco útil ya que podríamos, sin más, haber añadido esa funcionalidad a la clase original *ClsCalculo*, evitándonos el mantenimiento de dos clases diferentes. Vamos ahora a explorar la verdadera potencia del concepto de *Herencia*, y es la posibilidad de reemplazar los métodos y propiedades originales por nuevas implementaciones que den lugar a resultados distintos: en este caso, la nueva clase, cuando el método **Average** sea llamado, borrará todos los elementos de la matriz para quedarse en blanco. De esta forma, tendremos dos clases con diferente funcionalidad partiendo de una base común: los objetos de la clase *padre* acumulan números sin fin, mientras que en la segunda cada llamada a **Average** los limpia y comienza de cero.

La clase *ClsCalculo2* queda ahora como sigue:

```
' Gambas class file
INHERITS ClsCalculo
```

```
PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END

PUBLIC FUNCTION Average() AS Float

  DIM Nm AS Float
  DIM Vl AS Float
  DIM Total AS Integer

  IF SUPER._Numbers.Count = 0 THEN RETURN 0

  FOR EACH Vl IN SUPER._Numbers
    Nm += Vl
    Total += 1
  NEXT

  SUPER._Numbers.Clear()

  RETURN Nm / Total

END
```


Ahora existe un método *Average* dentro de *ClsCalculo2*, y este método reemplaza en la clase *hija* el método *Average* del *padre*. Cuando llamemos al método *Average* de un objeto de la clase *ClsCalculo2*, el intérprete no llamará a la función original del *padre*, sino a la implementación de la clase *hija*.

Modificamos la función **Main()** de pruebas con este nuevo código para comprobar el resultado:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo2

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()

  Calculo.Add(17.5)
  Calculo.Add(31.8)

  PRINT "Elementos: " & Calculo.Count & " - Media: " &
  Calculo.Average()

END
```

Como resultado, obtenemos en consola dos líneas que muestran el funcionamiento de la nueva clase:

```
Elementos: 3 - Media: 14.6
Elementos: 2 - Media: 24.65
```

Si en la primera línea de **Main()** definimos ahora que el objeto **Calculo** pertenece a la clase *padre* **ClsCalculo**, eliminamos el uso de *Count*, que no existe en el padre, y volvemos a ejecutar el programa, observaremos claramente la diferencia:

```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo

  Calculo.Add(12.5)
  Calculo.Add(23.8)
  Calculo.Add(7.5)

  PRINT "Media: " & Calculo.Average()

  Calculo.Add(17.5)
  Calculo.Add(31.8)

  PRINT "Media: " & Calculo.Average()

END
```


En este caso, se calcula primero la media de los tres primeros números y luego la media de los cinco números:

```
Media: 14.6
Media: 18.62
```

Podemos aprovecharnos más de la herencia, empleando la palabra *SUPER* que vimos anteriormente. La clase original *ClsCalculo* ya tenía el código necesario para calcular la media, y en la clase *hija ClsCalculo2* tan sólo necesitamos borrar los elementos de la matriz una vez calculada la media.

Podemos, entonces, modificar el código de la nueva función, **Average**, para que llame a la original y después borre los elementos, ahorrando, como siempre, tiempo y código:

```
' Gambas class file
INHERITS ClsCalculo
```

```
PROPERTY READ Count AS Integer

PRIVATE FUNCTION Count_Read() AS Integer

  RETURN SUPER._Numbers.Count

END

PUBLIC FUNCTION Average() AS Float

  DIM Nm AS Float

  Nm = SUPER.Average()
  SUPER._Numbers.Clear()

  RETURN Nm

END
```


La nueva función Average ahora se limita a llamar a la original, implementada en la clase *padre*, a almacenar ese valor, borrar los elementos de la matriz y devolver el valor.

Reemplazando métodos especiales: _New y _Free

Vamos a crear una nueva clase *hija* de *ClsCalculo*, que se comporte como la original, pero que, en este caso, en el momento de inicializarse un objeto de la clase, almacene unos valores. Pongamos de ejemplo una toma de muestras de temperatura.

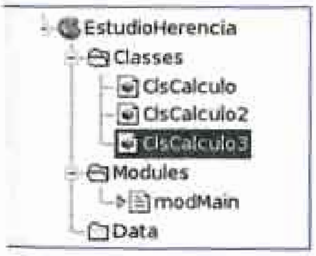


Figura 5. Nueva clase hija *ClsCalculo3*.

Los objetos, al inicializarse, pueden tomar los valores de temperaturas de la semana anterior de forma automática para que, después, el programador introduzca los de la semana en curso y se obtenga la media de la quincena. Para ello creamos la clase *ClsCalculo3*, con este código:

```
' Gambas class file
INHERITS ClsCalculo

PUBLIC SUB _New()

  SUPER.Add(12.5)
  SUPER.Add(15.3)
  SUPER.Add(18.4)
  SUPER.Add(19.12)
  SUPER.Add(21.15)
  SUPER.Add(20.4)
  SUPER.Add(19.5)

END
```

Igualmente escribimos y ejecutamos el código de Main() para comprobar el resultado:


```
PUBLIC SUB Main()

  DIM Calculo AS NEW ClsCalculo3

  Calculo.Add(29.2)
  PRINT "Media: " & Calculo.Average()

END
```

El resultado de la ejecución es:

```
Media: 19.44625
```

Como era de esperar, por lo explicado anteriormente, se ha ejecutado el método _New() de nuestra clase *hija*. Ahora bien, en este método tan sólo añadimos elementos,

pero no inicializamos la matriz, por lo que de haberse comportado el intérprete como en los casos anteriores, habríamos obtenido un error en tiempo de ejecución, al tener en el inicio la matriz con valor *Null*.

Lo que ha ocurrido en realidad, es que al reemplazar el método especial _New(), el intérprete ejecuta siempre el método original de la clase *padre* y luego la nueva implementación del *hijo*.

El método especial *_New()* no se reemplaza en las clases *hijas* como en el resto de métodos o propiedades, en su lugar se llama siempre al método original *_New()* del *padre* para que inicialice todo lo necesario y, después, si existe, se llama al método *_New()* de la clase *hija*. De este modo, el objeto siempre está listo e inicializado cuando se comienza a trabajar con él.

Añadamos ahora un destructor **_Free()** personalizado para nuestra clase heredera, en el cual indicamos el valor de la variable **_Numbers**:

```
' Gambas class file
INHERITS ClsCalculo

PUBLIC SUB _New()

  SUPER.Add(12.5)
  SUPER.Add(15.3)
  SUPER.Add(18.4)
  SUPER.Add(19.12)
  SUPER.Add(21.15)
  SUPER.Add(20.4)
  SUPER.Add(19.5)

END
```

```
PUBLIC SUB _Free()

  if SUPER._Numbers=NULL then print "NULO"

END
```

La ejecución del programa da ahora como resultado lo que vemos a continuación:

```
Media: 19.44625
NULO
```

Con esto se demuestra que el intérprete, como en el caso de *_New*, ha llamado primero al método *_Free* de la clase original, para llamar luego al método *_Free* de la clase *hija*.

Si nos ha surgido alguna duda respecto al orden de ejecución de algunas funciones, recordemos que siempre podremos ejecutar el código paso a paso para comprobarlo.

Limitaciones

La principal limitación de la herencia en Gambas, es que una clase no puede ser *hija* de otra que, a su vez, era *hija* de una tercera.

En otras palabras, sólo hay un nivel de herencia y en nuestros ejemplos anteriores no podríamos crear una clase *ClsCalculo4* que proviniera de *ClsCalculo2* o *ClsCalculo3*, puesto que éstas están empleando ya la palabra clave *INHERITS* para indicar que heredan las propiedades de *ClsCalculo*.

Esto implica, por ejemplo, que no podemos crear clases que sean herederas de *Form*, pues la clase *Form* proviene de *Window*.

8. 3 Crear un nuevo control con Gambas

Planteando el código

Vamos a examinar la posibilidad de crear nuevos controles gráficos directamente desde Gambas, aprovechando el concepto de herencia.

Crear controles nuevos programados en Gambas aporta dos ventajas fundamentales: su desarrollo es rápido al trabajar en un lenguaje de alto nivel, y el mismo código sirve tanto si programamos con *gb.gtk*, como si lo hacemos con *gb.qt*, ahorrándonos dos implementaciones para un mismo control con diferentes *toolkits*.

Las desventajas son una velocidad menor del código al ser interpretado, cuestión que tendrá más o menos importancia según la función del nuevo control, así como menor flexibilidad a la hora de moldear el control, dado que trabajaremos sobre los controles previamente aportados con Gambas y no sobre un *toolkit* para C o C++ como GTK+ o QT, ni a bajo nivel con X-Windows.

Para crear un control con Gambas, tenemos que partir de uno que exista ya. Si estamos acostumbrados a trabajar con otros entornos RAD de BASIC o C++, nos será familiar partir de un control *plantilla* con unas propiedades y métodos básicos para crear uno nuevo.

Los controles creados con Gambas han de partir de un control ya existente y creable, a su vez. No se puede partir directamente de la clase base *Control*, es decir, un control creado con *INHERITS CONTROL* no funcionará, ya que *Control* no tiene una representación gráfica, sino que es simplemente la base que define las propiedades, métodos y eventos mínimos para otros controles reales (*Label, Button*...).

Nuestro nuevo control será una etiqueta a la que llamaremos *ColorLabel*, similar al *Label*, pero que muestra el texto con un gradiente de color.

Para dibujar texto en diferentes colores, trazaremos cada letra por separado, desde un color inicial *Color1* hasta un color final *Color2*, utilizando un algoritmo algo rudimentario: tomamos la diferencia de valor de los dos colores y vamos aumentando el valor del color actual por cada nueva letra que dibujamos.

Con el fin de trazar cada letra por separado, tendremos que usar la clase *Draw* sobre un control base que será *DrawingArea*.

Implementación básica

Creamos un nuevo proyecto gráfico, que llamaremos *ColorLabel*, para nuestras pruebas. Dentro de él, originaremos un formulario *FMain*, también para nuestras pruebas, así como una clase llamada *ColorLabel* para implementar el control. Al igual que una etiqueta normal, definiremos una propiedad **Text** con el texto a mostrar, y añadiremos dos propiedades nuevas **Color1** y **Color2** que contienen el valor inicial y final del gradiente. Para manejar las tres propiedades, tendremos tres variables privadas: **hText** para el texto, **hColor1** para el primer color y **hColor2** para el segundo color.

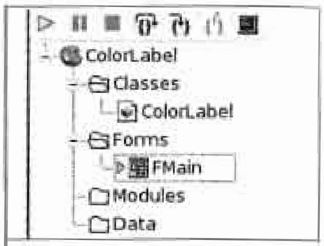


Figura 6. Nuevo proyecto gráfico *ColorLabel*.

Empezamos, pues, a escribir el código de la clase:

```
' Gambas class file
INHERITS DrawingArea

PRIVATE hText AS String
PRIVATE hColor1 AS Integer
PRIVATE hColor2 AS Integer

PROPERTY Text AS String
PROPERTY Color1 AS Integer
PROPERTY Color2 AS Integer
```

Implementar las funciones de lectura de las tres propiedades es trivial, basta con devolver los valores de nuestras variables privadas (recordemos que estamos escribiendo el código siempre dentro de la clase *ColorLabel*):

```
PRIVATE FUNCTION Color1_Read() AS Integer

  RETURN hColor1

END


PRIVATE FUNCTION Color2_Read() AS Integer

  RETURN hColor2

END

PRIVATE FUNCTION Text_Read() AS String

  RETURN hText

END
```

En cuanto a las funciones de escritura, hemos de asignar el valor que el usuario pasa a nuestras propiedades, y hemos de actualizar el control cada vez que una de ellas cambia, tarea que realizaremos en una función llamada **Redraw**(), aún por definir:

```
PRIVATE SUB Text_Write(Vl AS String)

  hText = Vl
  Redraw()

END
```

```
PRIVATE SUB Color1_Write(Vl AS Integer)

  hColor1 = Vl
  Redraw()

END

PRIVATE SUB Color2_Write(Vl AS Integer)

  hColor2 = Vl
  Redraw()

END
```

Llega el punto de implementar **Redraw()**, para lo cual recorreremos carácter por carácter la cadena almacenada en la variable **hText** e iremos representando letra por letra usando los distintos colores del gradiente. Para calcular la posición en el eje X de cada letra, nos servimos del método **Font.Width()** que determina el ancho de cada letra con la fuente actual del control:


```
PRIVATE SUB Redraw()

  DIM xPos AS Integer
  DIM Bucle AS Integer
  DIM hColor AS Integer
  DIM hDiff AS Integer

  ME.Clear
  hColor = hColor1
  IF Len(hText) THEN hDiff = (hColor2 - hColor1) /
  Len(hText)

  Draw.Begin(ME)
```

```
    FOR Bucle = 1 TO Len(hText)

      Draw.ForeColor = hColor
      Draw.Text(Mid(hText, Bucle, 1), xPos, 0)
      xPos = xPos + SUPER.Font.Width(Mid(hText, Bucle, 1))
      hColor = hColor + hDiff

    NEXT
    Draw.End()

  END
```

Finalmente, nos interesa que el control se encargue por sí mismo del redibujado de la ventana. Por defecto, todo lo dibujado sobre un *DrawingArea* desaparece si la ventana se oculta tras otra ventana o se minimiza. Para evitarlo, utilizaremos la propiedad **Cached** de *DrawingArea*, que se encarga de mantener un buffer y redibujar las partes expuestas al usuario del control:


```
  PUBLIC SUB _New()

    SUPER.Cached = TRUE

  END
```

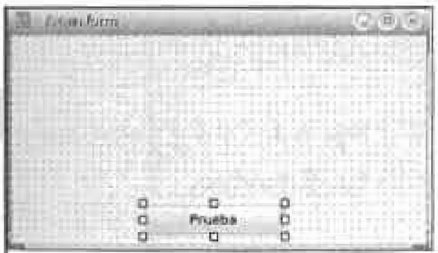


Figura 7. Texto de prueba sobre el formulario *FMain*.

Probemos ahora el control. Nuestro formulario *FMain* tendrá unas dimensiones de 225x240 píxeles y contendrá un botón llamado *BtnPrueba* con el texto **Prueba**.

El código del formulario consiste en crear un control **ColorLabel** en tiempo de ejecución, situándolo en el formulario con un texto de ejemplo y dos colores definidos para el gradiente:

```
PUBLIC SUB BtnPrueba_Click()

  DIM hLabel AS ColorLabel

  hLabel = NEW ColorLabel(ME)
  hLabel.Font.Size = 18
  hLabel.Move(0, 0, ME.Width, 100)
  hLabel.Color1 = Color.Blue
  hLabel.Color2 = Color.DarkBlue
  hLabel.Text = "Hola desde el ColorLabel"

END
```


Al ejecutar el programa y pulsar el botón, podremos ver ya nuestro control en funcionamiento. Ha sido una tarea rápida y sencilla, si bien se puede depurar bastante el código del control. Por ejemplo, este control tendrá problemas con caracteres con código ASCII mayor al 127, ya que en la codificación de Gambas (UTF-8) ocupan más de un byte, rompiendo el algoritmo que recoge el texto letra por letra.

Se puede solucionar convirtiendo la cadena a una codificación que emplee siempre un byte por letra (por ejemplo, ISO-8859-1) y reconvertirlo a UTF-8 a la hora de representarlo. También podríamos incluir una propiedad para alinear el texto a derecha, izquierda o centro, así como mejorar el algoritmo para obtener el gradiente, separando los componentes RGB de cada color, tareas todas ellas que dejamos si estamos interesados en practicar.



Figura 8. Control *ColorLabel* en funcionamiento.

Este nuevo control se puede reutilizar en todos los programas que diseñemos añadiendo la clase *ColorLabel* al proyecto.

8. 4 Nuevos componentes con Gambas

Cuando abrimos las propiedades de un proyecto y pulsamos la pestaña **Componentes**, observamos los módulos adicionales que podemos emplear en un programa escrito con Gambas, los cuales aportan nuevas funcionalidades.

Desde Gambas, tenemos también la posibilidad de crear nuevos componentes. Estos pueden ser de cualquier índole, aunque en el próximo ejemplo será un componente que aporta un control adicional: el *ColorLabel* que creamos en el apartado anterior.

Si bien vamos a crear aquí un nuevo componente gráfico, los componentes se limitan a aportar nuevas clases que no tienen por qué estar relacionadas con la interfaz gráfica.



Figura 9. Componentes del proyecto.

□□□□□ Preparación del código. Palabra clave EXPORT

Dentro de un programa Gambas, tendremos diversos módulos, clases y formularios. Implementar un control en Gambas supone tan sólo crear un programa que exportará algunas de sus clases. Preparar un programa Gambas para ser un componente, simplemente implica definir qué clases serán visibles y cuáles no.

Para indicar que una clase será visible desde el programa principal, se usa la palabra clave EXPORT al principio de la misma antes del resto del código. Por tanto, en nuestro programa *ColorLabel* indicamos esta palabra clave al inicio de la clase *ColorLabel*:

```
' Gambas class file
EXPORT
INHERITS DrawingArea

PRIVATE hText AS String
...
```


Ahora es necesario compilar el programa para tener un ejecutable con la nueva clase exportada.

□□□□□ Archivos necesarios, ubicaciones

Cada componente Gambas necesita, en primer lugar, del programa en sí, que tendrá siempre como nombre *gb.x.gambas*, donde *x* será un nombre que nosotros elijamos.

A partir de ahora, vamos a llamar a nuestro nuevo componente *controles* y, por tanto, el nombre del componente dentro del sistema de archivos será *gb.controles.gambas.*

Es necesaria también una ubicación específica para dicho archivo. Todos los componentes de Gambas han de estar situados en una carpeta que, salvo que hayamos compilado Gambas desde los fuentes empleando una opción - -*prefix* específica, será */usr/lib/gambas2*.

Si hemos compilado Gambas usando una opción --*prefix* distinta de */usr* en el script *configure* (por ejemplo, */usr/local*), tendremos que buscar las rutas indicadas a partir de la que seleccionamos, como */usr/local/lib/gambas2* en este caso.

Copiaremos nuestro ejecutable a dicha carpeta, para lo cual, desde la consola y como *root*, entraremos en la carpeta del proyecto y copiaremos el ejecutable con el nuevo nombre (por supuesto, si lo preferimos, podemos hacerlo con Konqueror o Nautilus):

```
cp ColorLabel.gambas /usr/lib/gambas2/gb.controles.gambas
```

Los componentes en Gambas necesitan también de un archivo con extensión **.component** para ser reconocidos como tales. Vamos a crear este archivo a mano con cualquier editor. Nosotros vamos a usar **gedit**, pero puede ser cualquiera (recordemos que debemos ser root):

```
gedit /usr/lib/gambas2/gb.controles.component
```

El contenido de dicho fichero incluye varios parámetros que se exponen a continuación:

```
[Component]
Key=gb.controles
Name=Additional controls
Author=Daniel Campos
Group=Adicionales
Controls=ColorLabel
Require=gb.qt
```

La clave [Component] siempre debe estar al inicio del archivo. En el valor Key hay que situar el nombre del componente sin extensión (en este caso, el fichero *gb.component.gambas* que hemos copiado, pero sin la extensión *gambas*). En la clave Name

se indica el nombre del componente que aparecerá luego como descripción a la hora de elegir los componentes de un proyecto. En **Author** situamos el nombre del autor del componente.

Después vienen una serie de claves opcionales: **Group** sólo se usa si queremos añadir controles gráficos y supone la pestaña de controles donde queremos que se añada el nuevo. Lo mismo se aplica para **Controls**, que se referirá a cada uno de los controles que queremos que aparezcan en dicha pestaña. **Require** se utiliza si nuestro componente depende de otros, en este caso, de **gb.qt**. Si hubiéramos compilado el proyecto para *gb.gtk*, tendría que ser ésa la dependencia indicada.

Una vez creado y guardado el archivo *gb.controles.component*, tenemos finalmente que copiar dos archivos dentro de */usr/share/gambas2/info*, que proporcionan información adicional para el intérprete de Gambas.

Dentro de nuestra carpeta del proyecto se han creado también tras compilar, dos archivos llamados **.list** e **.info**. Si deseamos verlos, recordemos que los archivos que comienzan por un punto son ocultos, por tanto, hemos de usar *ls -a* desde la consola o bien activar la opción de ver archivos ocultos desde Konqueror o Nautilus. Dichos archivos han de copiarse a su ubicación definitiva con el nombre del componente (en este caso, **gb.controles.info** y **gb.controles.list**):

```
cp .info /usr/share/gambas2/info/gb.controles.info
cp .list /usr/share/gambas2/info/gb.controles.list
```

Probando el nuevo componente

Tras estos pasos cerraremos el IDE de Gambas si estaba abierto y lo ejecutaremos de nuevo, creando un proyecto gráfico llamado *MiPrueba* en el que añadiremos un formulario.

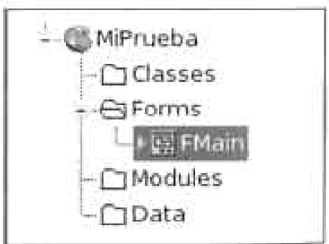


Figura 10. Formulario *FMain* sobre el proyecto *MiPrueba*.

Dentro de **Proyecto | Propiedades**, seleccionaremos la pestaña de **Componentes**.

Buscaremos y marcaremos el componente *gb.controles* y pulsaremos Aceptar para tenerlo como dependencia.



Figura 11. Selección del componente *gb.controles* como dependencia.



Figura 12. Pestaña **Adicionales** de la ventana de selección.

Ahora, en la ventana de selección de controles aparece una nueva pestaña **Adicionales**, tal y como habíamos indicado en el archivo *component* (**Figura 12**).

Al situarnos en esta nueva pestaña, aparecen los controles disponibles. Desgraciadamente, en este momento no se contempla la posibilidad de que esos controles tengan un icono propio, por lo que aparecen sin un dibujo representativo (**Figura 13**).

No obstante, al situarnos sobre él con el ratón, se muestra el botón y el *ToolTip* indicándonos el nuevo control.

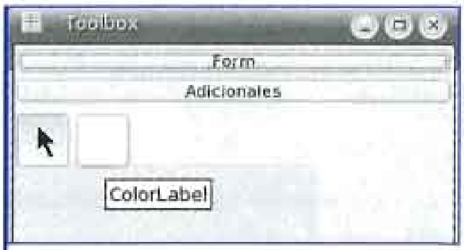


Figura 13. Controles **Adicionales**.

Como hacemos con el resto de controles, lo arrastraremos al formulario y lo situaremos como deseemos.



Figura 14. Control *ColorLabel1*.

Finalmente, en el código del formulario indicamos el texto y los colores de la etiqueta:

```
PUBLIC SUB Form_Open()

  ColorLabel1.Font.Size = 16
  ColorLabel1.Text = "Hola desde el componente"
  ColorLabel1.Color1 = Color.Blue
  ColorLabel1.Color2 = Color.Black

END
```

Al ejecutarlo, observamos el resultado. Ya disponemos de un nuevo componente reutilizable en muchos proyectos y que se puede distribuir entre distintos equipos siguiendo los pasos indicados.

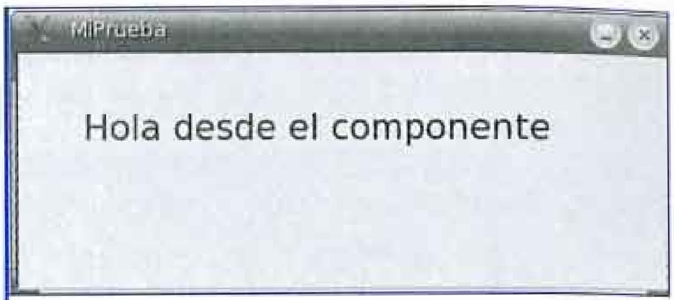


Figura 15. Resultado del nuevo componente.

Un componente escrito en Gambas es tan sólo un programa que exporta clases, por tanto, se puede depurar fácilmente haciendo las pruebas como en cualquier otro proyecto para, después, distribuirlo como componente.

9. Acceso a la API

9. Acceso a la API
9. Acceso a la API
9. Acceso a la API

En realidad, el título *Acceso a la API* está tomado de los programadores de Visual Basic™, en los cuales es muy frecuente realizar llamadas a funciones externas, alojadas en librerías también externas escritas en lenguaje C. Tal vez, en este caso no sea correcto, dado que Linux es tan sólo un núcleo y el resto de librerías (incluyendo la estándar de C, *glibc*) no son propiamente la API de este sistema operativo, sino campos de utilidades que se añaden al sistema. Pero, de todas formas, sirve de introducción para el asunto que vamos a tratar.

Un sistema GNU/Linux puede contar con librerías distintas de otro, pero en la práctica siempre vamos a encontrar como mínimo la librería de C (*glibc*), posiblemente una librería para diseño de interfaces de texto (*ncurses*) y, si hay un sistema gráfico instalado, las librerías de X-Window y otras como *glib*, *gtk+*, *atk*, etc. Al margen

de éstas, todo sistema suele tener instaladas librerías como *libcurl*, para acceso a diversos protocolos de red, o *libaspell*, para corrección ortográfica.

Desde Gambas podemos sacar provecho de las funciones aportadas por estas librerías, declarando funciones externas e indicando la librería en la que se deben buscar. El programa aporta, además, ciertos elementos para la gestión de memoria y punteros. Las librerías accesibles serán aquellas escritas en lenguaje C, que sigan el estándar definido (ABI), o bien escritas en otros lenguajes si respetan dicho estándar (por ejemplo, escritas con FreePascal). No es posible, en este momento, acceder a funciones de librerías escritas en C++, al menos de forma directa.

Si bien en sistemas Windows™ es muy habitual recurrir a este modo de trabajo, hay que matizar que en Linux muchas tareas se pueden resolver simplemente llamando a procesos externos, labor más sencilla, por lo general, y menos problemática en la depuración de los programas. Por tanto, recurrir a la gestión de procesos cuando sea posible, en lugar de a este sistema, es, en general, una buena idea.

Hay que tener en cuenta que la llamada a funciones externas, tanto desde Gambas como desde otros lenguajes interpretados (Mono, por ejemplo), requiere un trabajo de conversión de tipos, entre otras tareas, lo cual consume un tiempo de procesador. Tratar, pues, de emplear llamadas a funciones con el fin de incrementar la velocidad del programa, puede no ser una buena idea en todos los casos.

9.1 Declarar una función externa

La declaración se debe realizar al principio de una clase o módulo, del mismo modo que las variables globales. Al principio se declarará la función como *PUBLIC* o *PRIVATE*, para definir su ámbito, limitado a la clase donde se aloja o a todo el programa. Por defecto son privadas.

A continuación, la palabra clave **EXTERN** indica que vamos a declarar una función externa, seguida por su nombre, que será el nativo de la función en la librería.

Hay que tener en cuenta que C sí distingue entre mayúsculas y minúsculas y, por tanto, tratar de declarar *Printf* en lugar de *printf* para la librería *glibc*, dará como resultado un error.

Como en el resto de funciones de Gambas, se indicarán los parámetros que reciben entre paréntesis, y después el tipo de dato devuelto. Opcionalmente, se puede indicar al final la librería en que se encuentra la función, así como un *alias*, o nombre alternativo, a utilizar desde este programa, para llamar a la función, que se empleará, por ejemplo, en ciertas funciones cuyo nombre coincide con palabras clave del lenguaje Gambas. Supongamos la función **strcmp** de la librería estándar de C, que se define en este lenguaje como:

```
int strcmp(const char *s1, const char *s2);
```

La declaración en Gambas será:

```
EXTERN strcmp(s1 AS STRING,s2 AS STRING) AS INTEGER IN
"libc:6"
```

□□□□□ Cómo denominar la libreria

Tras la palabra clave IN, se presenta entre comillas el nombre de la librería. En principio, se puede indicar un nombre simple, como *libc* o *libgstreamer*. En ese caso, Gambas tratará de encontrar la versión más moderna instalada en el sistema. A veces, sin embargo, puede interesarnos una versión concreta de una librería, en cuyo caso se reflejará indicando el símbolo : tras el nombre y, a continuación, el número de versión, por ejemplo *libgl:1.0.7667* o *libc:6*.

En sistemas GNU/Linux, dentro de las carpetas */lib* o */usr/lib* encontraremos la mayoría de las librerías que hay en nuestro sistema. A partir de esos nombres de archivo, podemos deducir el nombre de la librería, por ejemplo:

```
/usr/lib/libglib-2.0.so.0.600.4    ->    "libglib-
2.0:0.600.4"
```

Como podemos observar, el número de versión es siempre la serie de números que se encuentra tras el texto **.so.** en el nombre de archivo, y que todo lo que viene antes forma parte del nombre de la librería. En este caso, el valor **2.0** tras el texto **libglib-** es parte del nombre y no de la versión. Especificar un número de versión puede ser problemático cuando se trata de instalar un programa entre diferentes equipos con distintas distribuciones GNU/Linux instaladas, que pueden tener versiones ligeramente diferentes de estas librerías o cambiar tras una actualización del sistema. Por tanto, hay que aplicar esta característica sólo en casos imprescindibles, o tal vez emplear el número mayor y dejar el menor y la revisión a elección de Gambas.

Si en un módulo o clase se han de definir varias funciones de una misma librería, no es necesario especificarla en todas las declaraciones, basta con indicarla como librería por defecto con la palabra clave **LIBRARY** y Gambas la empleará en todas las declaraciones en las cuales no se especifique la librería.


```
LIBRARY "libglib-2.0"

EXTERN g_utf8_strlen(p AS String, Mx AS Integer) AS
Integer
EXTERN g_utf8_strup(Buf AS String, Mx AS Integer) AS
Pointer
EXTERN getchar() AS Integer IN "libc:6"
```

Aquí, las dos primeras funciones pertenecen a **libglib**, y la tercera a **libc**.

El uso de los alias

Supongamos la función *free()* de la librería estándar de C, que se emplea para liberar memoria asignada, por ejemplo, con *malloc()*. El problema para declararla en Gambas es que ya existe *Free* como palabra clave reservada, por tanto, tendremos que indicar un nombre alternativo:

```
EXTERN FreePointer(Ptr AS Pointer) IN "libc:6" EXEC
"free"
```

A la hora de escribir código Gambas, llamaremos siempre a la función FreePointer, que internamente llamará a la original *free* de la librería de C.

Tipos de datos

Existe una correspondencia entre los tipos de datos básicos de Gambas y los de C. Hay que tener cuidado con los *falsos amigos*, un tipo *long* de C, no es un tipo *Long* de Gambas, sino un tipo *Integer*; y un tipo *float* de C no es el *Float* de Gambas, sino el *Single*. Las siguientes correspondencias son aplicables a Gambas sobre GNU/Linux 32 bits/Intel.

Correspondencia de datos entre C y Gambas

C	Gambas	C	Gambas
char	*Byte*	*long long*	*Long*
short	*Short*	*float*	*Single*
int, long	*Integer*	*double*	*Float*

El tipo *Pointer* de Gambas se emplea para definir un puntero. En Gambas-GNU/Linux-32 bits es exactamente igual a *Integer*, pero en otras arquitecturas podría cambiar, por lo cual es recomendable utilizar siempre el tipo de dato *Pointer* en lugar de *Integer* para referirse a punteros. El tipo de datos *String* sólo debe emplearse si la cadena a pasar como parámetro no va a ser modificada o reasignada por la función de C, en otro caso debe utilizarse un dato tipo *Pointer*. Por último indicar que es posible pasar datos tipo *Object*, pero no está permitido usar tipos *Variant*.

9. 2 Funciones auxiliares

Gambas aporta algunas utilidades para trabajar con llamadas a funciones.

```
Puntero = Alloc ( Tamaño AS Integer [ , Cantidad AS
Integer ] )
```

La función **Alloc** es similar a la homónima de C: reserva un bloque de *Tamaño*Cantidad bytes*. En este caso, el intérprete de Gambas lleva, además, un contador de las asignaciones y liberaciones de memoria realizadas, indicando, con un mensaje de advertencia al final del programa, si quedó memoria sin liberar.

```
Nuevo Puntero = Realloc ( Antiguo puntero AS Pointer ,
Tamaño AS Integer , Cantidad AS Integer )
```

Realloc toma un puntero, previamente asignado con *Alloc* o *Realloc*, y cambia su tamaño, manteniendo los datos almacenados; devuelve un puntero a la nueva área de memoria reasignada; y, al igual que *Alloc*, mantiene la cuenta de asignaciones/liberaciones de memoria.

```
Free ( Puntero )
```

Free libera un puntero asignado con *Alloc* o *Realloc*, de igual modo que la función homónima de C, pero manteniendo la cuenta de asignaciones y liberaciones de memoria.

Muchas librerías aportan sus funciones propias para asignar o liberar memoria. Puede ser necesario que empleemos estas funciones en lugar de *Alloc* o *Free*, si así lo requiere el programa, para mantener la coherencia del código y funcionalidad de la librería.

```
Cadena = StrPtr ( puntero AS Pointer )
```

Cuando una cadena ha de tratarse como un puntero, esta función nos permite obtener una copia como tipo de dato *String*, siempre y cuando **puntero** apunte a una cadena terminada en carácter *\0* (nulo), que es lo habitual cuando se trabaja con C.

Gambas permite escribir y leer en memoria como si fuese otro flujo cualquiera. Así, utilizando un puntero como parámetro, se pueden utilizar las instrucciones

habituales en archivos: *READ*, *WRITE*, etc. Entre otras posibilidades ofrecidas por esta solución, éste es el modo actual en el que el programador puede acceder a los datos de una estructura de C.

Está previsto que la versión 2 de Gambas añada el componente *gb.api*.

Este componente se está desarrollando para dar soporte directo al trabajo con estructuras de datos y a retrollamadas, que aumentarán y simplificarán las capacidades de Gambas para manejar librerías externas.

9. 3 Un ejemplo con libaspell

Aspell proporciona una API para corrección de texto, permite analizar palabras, indicar si son válidas o no según el diccionario empleado, y ofrecer palabras alternativas a las consideradas incorrectas, entre otras capacidades.

Disponemos de más información sobre *libaspell* en su página oficial:
http://aspell.sourceforge.net/

Para poder utilizar este código, tendremos que instalar *Aspell* en el sistema, así como el diccionario de nuestro idioma.

Por ejemplo, en un sistema gnuLinex 2004 en castellano (basado en Debian), los paquetes instalados en el sistema son:

aspell
aspell-bin
libaspell-dev
aspell-es
libaspell15

Crearemos un nuevo proyecto gráfico llamado **Aspell**, con un único formulario **FMain** en el cual se contendrá un botón llamado **BtnCheck**, un *TextBox* llamado **TxtIn**, un *ListBox* llamado **Lista** y un *ProgressBar* llamado **Pbar**.

Al inicio del código del formulario definimos las funciones de **libaspell**.



Figura 1. Formulario del proyecto *Aspell*.

```
LIBRARY "libaspell"

  EXTERN new_aspell_config() AS Pointer
  EXTERN aspell_config_replace(Cfg AS Pointer, Var AS
  String, Value AS String)
  EXTERN new_aspell_speller(Cfg AS Pointer) AS Pointer
  EXTERN aspell_error_number(Err AS Pointer) AS Integer
  EXTERN aspell_error_message(Err AS Pointer) AS
  Pointer
  EXTERN to_aspell_speller(Err AS Pointer) AS Pointer
  EXTERN aspell_speller_check(V1 AS Pointer, V2 AS
  String, V3 AS Integer) AS Integer
  EXTERN aspell_speller_suggest(V1 AS Pointer, V2 AS
  String, V3 AS Integer) AS Pointer
  EXTERN aspell_word_list_elements(V1 AS Pointer) AS
  Pointer
  EXTERN aspell_string_enumeration_next(Ptr AS Pointer)
  AS Pointer
  EXTERN delete_aspell_string_enumeration(Ptr AS
  Pointer)
  EXTERN delete_aspell_config(Cfg AS Pointer)
```


Al pulsar el botón BtnCheck, realizaremos el proceso de análisis del texto introducido en el cuadro TxtIn, para ir mostrando el resultado en la lista.

```
PUBLIC SUB BtnCheck_Click()

  DIM spell_config AS Pointer
  DIM possible_err AS Pointer
  DIM spell_checker AS Pointer
  DIM suggestions AS Pointer
  DIM elements AS Pointer

  DIM MyStr AS String[]
  DIM Bucle AS Integer
  DIM BufErr AS String
  DIM Sum AS Float
  DIM hPtr AS Pointer
  DIM sPtr AS String
```

En primer lugar, dividimos el texto introducido por el usuario en palabras, separadas por espacios.


```
  PBar.Value = 0
  Lista.Clear()
  MyStr = Split(TxtIn.Text, " ")
  Sum = 1 / MyStr.Count
```

Realizamos las primeras llamadas a la librería *libaspell*. Para ello tomamos una configuración por defecto y aplicamos dos atributos: el texto estará codificado como UTF-8 y se usará el diccionario que corresponda con el lenguaje del sistema.

Aplicamos la configuración y, de existir un error (por ejemplo, que el diccionario no se encuentre disponible), indicamos el mensaje correspondiente:

```
spell_config = new_aspell_config()
aspell_config_replace(spell_config, "lang",
Application.Env["LANG"])
```

```
aspell_config_replace(spell_config, "encoding",
"utf-8")



possible_err = new_aspell_speller(spell_config)

IF aspell_error_number(possible_err) <> 0 THEN
  Message.Error(StrPtr(aspell_error_message
  (possible_err)))
  RETURN
ELSE
  spell_checker = to_aspell_speller(possible_err)
END IF
```


Después entramos en un bucle en el cual se analiza, palabra por palabra, el texto introducido. En caso de error, se dará una lista de sugerencias, pero si está todo correcto, se indicará con un *OK*.

```
FOR Bucle = 0 TO MyStr.Count - 1

    PBar.Value = PBar.Value + Sum
    MyStr[Bucle] = Trim(MyStr[Bucle])
    IF Len(MyStr[Bucle]) > 0 THEN
      IF aspell_speller_check(spell_checker,
      MyStr[Bucle], - 1) THEN
        lista.Add("OK    -> " & MyStr[Bucle])
      ELSE

        BufErr = " ( "
        suggestions = aspell_speller_suggest
        (spell_checker, MyStr[Bucle], - 1)
        elements = aspell_word_list_elements
        (suggestions)
```

```
            DO WHILE TRUE
              hPtr = aspell_string_enumeration_next(elements)
              IF hPtr = 0 THEN
                BREAK
              ELSE
                sPtr = ""
                INPUT #hPtr, sPtr
                BufErr = BufErr & sPtr & " "
              END IF

            LOOP
            delete_aspell_string_enumeration(elements)

            BufErr = BufErr & ")"
            lista.Add("ERROR -> " & MyStr[Bucle] & BufErr)

          END IF
        END IF
        WAIT 0.001
      NEXT
```

Finalmente liberamos los elementos necesarios empleados en la corrección de texto.

```
      delete_aspell_config(spell_config)
      PBar.Value = 1

    END
```

9. 4 Obtener información acerca de la librería

Normalmente, las librerías aportan multitud de constantes y macros definidas en sus ficheros de cabecera. A la hora de trabajar con ellas debemos conocer sus valores, ya

que no se puede recurrir directamente a los **#include** de C. Veamos este ejemplo, que utiliza la constante GTK_WINDOW_TOPLEVEL:

```
#include <gtk/gtk.h>
int main(void)
{
        GtkWindow *win;

        gtk_init(0,0);

        win=gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_widget_show_all(win);

        gtk_main();
}
```


Con el fin de conocer el valor de esta constante, o cualquier otra aportada por una librería, podemos recurrir al menos a cuatro métodos:

1. Consultar la documentación de la propia librería, que en este caso (GTK+), es bastante completa. La podemos encontrar en la siguiente dirección: **http://developer.gnome.org/doc/API/2.0/gtk/gtk-Standard-Enumerations.html#GtkWindowType**.

2. Indicar que **GTK_WINDOW_TOPLEVEL** tiene el valor 0 como parte de la enumeración *GtkWindowType*.

3. Buscar en los ficheros de cabecera de la librería, que se suelen depositar a partir de la ruta */usr/include*. En el caso que tratamos en el ejemplo, este valor se encuentra en el fichero */usr/include/gtk-2.0/gtk/gtkenums.h*.

4. Compilar un pequeño programa en C que muestre el valor problemático que no alcanzamos a averiguar por otros sistemas.

```
#include <gtk/gtk.h>
#include <stdio.h>

void main(void)
{
  printf("%d\n",GTK_WINDOW_TOPLEVEL);
}
```

Podemos ya pasar el pequeño programa a Gambas. Para ello, creamos un programa de consola con un solo módulo *modMain* que contenga este código:

```
' Gambas module file
LIBRARY "libgtk-x11-2.0"

CONST GTK_WINDOW_TOPLEVEL AS Integer = 0

EXTERN gtk_init(Argv AS Pointer, Argc AS Pointer)
EXTERN gtk_main()
EXTERN gtk_window_new(wType AS Pointer) AS Pointer
EXTERN gtk_widget_show(wid AS Pointer)

PUBLIC SUB Main()

  DIM Win AS Pointer

  gtk_init(0, 0)

  Win = gtk_window_new(GTK_WINDOW_TOPLEVEL)
  gtk_widget_show(Win)

  gtk_main()

END
```

Si lo ejecutamos podemos observar que el resultado es equivalente al del programa en C.

9. 5 Resumen

En este momento, Gambas autoriza el acceso a funciones de librerías externas. La única condición es que sigan el ABI estándar. Las propias instrucciones del programa permiten reservar y liberar memoria. En el futuro, a través de *gb.api* (no detallado aquí, pues se encuentra en pleno desarrollo y su interfaz puede cambiar), la gestión de estructuras y retrollamadas será posible.

Pero no se recomienda el uso de esta funcionalidad, salvo que esté plenamente justificado. Gambas aporta componentes para tareas variadas que cubren buena parte de las necesidades que se encuentran en los programas habituales, y su capacidad de gestión de procesos externos lo hace apropiado para casi cualquier tarea. Trabajar con llamadas a funciones externas puede hacer el programa más difícil de depurar, sensible a los cambios en el sistema (actualizaciones de librerías, cambio de versión de la distribución...) y limita la portabilidad del código (por ejemplo, de un sistema GNU/Linux, donde se desarrolla, a una plataforma FreeBSD o, en el futuro, en la versión Windows de Gambas).

Marcas registradas

Los siguientes términos son marcas registradas en Estados Unidos u otros países:

- Linux es una marca registrada de Linus Torvalds.
- Debian es una marca registrada de Software in the Public Interest, Inc.
- SUSE es una marca registrada de SuSE AG.
- Mandriva es una marca registrada de Mandrakesoft S.A. y Mandrakesoft Corporation.
- GNOME es una marca registrada de la Fundación GNOME.
- KDE, K Desktop Environment, es una marca registrada de KDE e.V.
- Microsoft, Windows y Visual Basic son marcas registradas de Microsoft Corporation.
- gnuLinEx es una marca registrada de la Junta de Extremadura.
- Java y todos los nombres y logos basados en Java son marcas registradas de Sun Microsystems, Inc.
- UNIX es una marca registrada de The Open Group.
- Mozilla y Firefox son marcas registradas de The Mozilla Organization.
- Apple y Macintosh son marcas registradas de Apple Computer Corporation.

Otras empresas, productos y nombres de servicios pueden ser marcas registradas o servicios de otras compañías.

GAMBAS

Gracias a Gambas, el programador involucrado en el Software Libre, bien sea por su contribución a la comunidad o por sus necesidades profesionales, dispone ahora de un entorno de desarrollo que conjuga la sencillez y potencia necesarias para ejecutar aplicaciones complejas y bien estructuradas en un tiempo récord.

Gambas permite realizar todo tipo de aplicaciones profesionales, desde CGIs a complejas interfaces gráficas, las cuales no están atadas a una librería concreta (QT o GTK+), sino que pueden ser compiladas para funcionar indistintamente con ambas librerías sin tocar una sola línea de código.

Todos los programadores provenientes de entornos Windows™ encontrarán una herramienta en la que sentirse cómodos: un entorno RAD, un intérprete de BASIC Visual, depuración de programas paso a paso, sistema de ayuda integrado, creación de paquetes de distribución, soporte nativo de aplicaciones multi-idioma...y los programadores acostumbrados a explotar los conceptos modernos de programación hallarán la posibilidad de utilizar clases, objetos, herencia y otras características con una sencillez nunca antes vista en una herramienta libre.

Los analistas encontrarán, igualmente, una herramienta alternativa para ofrecer soluciones rápidas a los clientes específicos que ya trabajan con Software Libre, gracias a sus ventajas de estabilidad y economía a través de esta herramienta.

Los administradores de sistemas tienen ahora la posibilidad de crear scripts de administración con un lenguaje más potente y flexible que la shell Bash.

9 788493 437121

Avda. Portugal, 85-local
28011 Madrid
Tlf.: 91 577 03 55/18
Fax: 91 577 06 18

Edit Lin